51CTO.com
技术成就梦想

我们只谈开发

开发专刊

创刊号 2014年04月
总第034期

开发专刊 2014年04月 第34期 我们只谈开发

51CTO.com
技术成就梦想

开发专刊

主编的话

本次的专刊为大家提供了Oracle最新推出的Java SE 8详细的开发教程，从解读到探究Java 8最新特性，到进阶深入学习等等相关内容。

对于Java 8，最让开发者追捧的是Lambda表达式的加入，其本身是符合技术发展方向的，只是稍微来晚了些。通过引入Lambda，最直观的一个改进是，不用再写大量的匿名内部类。事实上，还有更多由于函数式编程本身特性带来的提升。

当然，Lambda的引入让集合操作也得到了极大的改善。 不过，对大多数Java程序员来说，他们最熟悉的内容是面向对象，函数式编程是个陌生的概念，是一种“全新”的思维模式。

下面，大家开始Java 8的全新特性之旅吧。

PS:专刊相关内容大部分来自互联网整理。

51CTO开发频道寄语

出版方:

北京无忧创想信息技术有限公司

责任编辑:

林师授

封面设计:

苍旭

联系方法:

邮箱：linss@51cto.com

电话：010-68476606-8123

出版日期：2014年04月30日

欢迎来稿，请发送邮件至

linss@51cto.com

目录 Contents

2014年04月
总第34期

编程排行

解读Java8

Java8特性探究

Java8 进阶学习

编者按

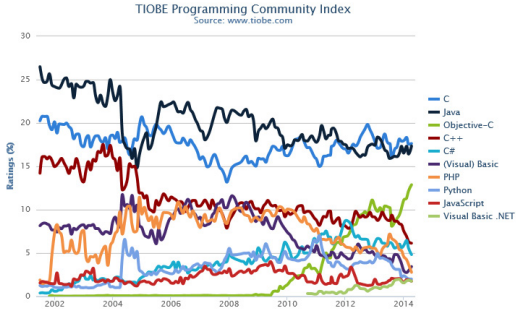
TIOBE社区今天发布了2014年4月的编程语言排行榜，官方已经不给本次的排行短评，除了Perl创下历史新低，其他编程语言基本没有什么变化。想喷的尽情来喷吧。

2014年4月编程语言排行榜：Perl 创历史新低

TIOBE社区今天发布了2014年4月的编程语言排行榜，官方已经不给本次的排行短评，除了Perl创下历史新低，其他编程语言基本没有什么变化。想喷的尽情来喷吧。

排行榜前20名：

Apr 2014	Apr 2013	Change	Programming Language	Ratings	Change
1	1		C	17.631%	-0.23%
2	2		Java	17.348%	-0.33%
3	4	↑	Objective-C	12.875%	+3.28%
4	3	↓	C++	6.137%	-3.58%
5	5		C#	4.820%	-1.33%
6	7	↑	(Visual) Basic	3.441%	-1.26%
7	6	↓	PHP	2.773%	-2.65%
8	8		Python	1.993%	-2.45%
9	11	↑	JavaScript	1.750%	+0.24%
10	12	↑	Visual Basic .NET	1.748%	+0.65%
11	10	↓	Ruby	1.745%	-0.23%
12	17	⇈	Transact-SQL	1.170%	+0.45%
13	9	⇊	Perl	1.027%	-1.31%
14	52	⇈	F#	0.966%	+0.83%
15	19	⇈	Assembly	0.853%	+0.14%
16	13	↓	Lisp	0.797%	-0.11%
17	18	↑	PL/SQL	0.782%	+0.07%
18	24	⇈	MATLAB	0.760%	+0.24%
19	15	⇊	Delphi/Object Pascal	0.746%	-0.09%
20	35	⇈	D	0.708%	+0.39%



21-50编程语言排名：

Position	Programming Language	Ratings
21	Logo	0.693
22	SAS	0.688
23	ML	0.643
24	Pascal	0.598
25	PostScript	0.578
26	OpenEdge ABL	0.539
27	ActionScript	0.438
28	PL/I	0.434
29	COBOL	0.429
30	Ladder Logic	0.380
31	ABAP	0.367
32	cT	0.360
33	C shell	0.352
34	Fortran	0.347
35	Lua	0.332
36	Go	0.324
37	Ada	0.287
38	Scratch	0.283
39	Emacs Lisp	0.269
40	R	0.265
41	Z shell	0.253
42	Groovy	0.247
43	S-PLUS	0.241
44	Io	0.229
45	Tcl	0.226
46	NXT-G	0.225
47	JScript.NET	0.220
48	Scala	0.219
49	Prolog	0.209
50	OCaml	0.197

(Visual) FoxPro, 4th Dimension/4D, Arc, ATLAS, Automator, Avenue, Awk, Bash, BlitzMax, Bourne shell, cg, CL (OS/400), Clean, Common Lisp, Erlang, Factor, Felix, Forth, Haskell, Icon, Inform, Informix-4GL, J, JavaFX Script, Korn shell, LabVIEW, M4, Magic, Max/MSP, Mercury, Modula-2, Moto, NATURAL, OpenCL, PILOT, Programming Without Coding Technology, Pure Data, Q, Revolution, RPG (OS/400), S, Scheme, Slate, Smalltalk, SPARK, Standard ML, TOM, VBScript, VHDL, X10

内容摘自：知乎

Java8引入Lambda表达式的利弊？

郑晔，程序员，乐于编写整洁代码

函数式编程是技术的发展方向，而Lambda是函数式编程最基础的内容，所以，Java 8中加入Lambda表达式本身是符合技术发展方向的。

通过引入Lambda，最直观的一个改进是，不用再写大量的匿名内部类。事实上，还有更多由于函数式编程本身特性带来的提升。比如：代码的可读性会更好、高阶函数引入了函数组合的概念。

此外，因为Lambda的引入，集合操作也得到了极大的改善。比如，引入stream API，把map、reduce、filter这样的基本函数式编程的概念与Java集合结合起来。在大多数情况下，处理集合时，Java程序员可以告别for、while、if这些语句。

随之而来的是，map、reduce、filter等操作都可以并行化，在一些条件下，可以提升性能。

不过，对大多数Java程序员来说，他们最熟悉的内容是面向对象，函数式编程是个陌生的概念，是一种“全新”的思维模式。对于喜欢墨守陈规的大多数而言，这无疑会增加Java的入门成本，以及向新版本迁移的成本。

还有一件事，Lambda本身是借助invokedynamic实现的，这是这个Java 7加入的新指令第一次在Java语言层面上得到应用。因为它的存在，我们在某种程度上可以绕过Java的类型系统，很难说这是好是坏。

Lambda表达式的短期目标是配合Collections Bulk Operation一起使用，使得Java具备很多其它编程语言具备的API，并将集合类的迭代内部化（以前是你自己iterate，现在是JDK内部给你iterate），以便施行诸如并行计算等灵活特性。

其长期目标在于促进Java语言进化，设计者认为函数式编程是一个不错的方向。也正是由于这个原因，我们没有简单的使用静态编译的内部类去实现Lambda，而是采用更灵活的invokedynamic。

Java 8 正式发布，新特性全搜罗

■ Java 8版本最大的改进就是Lambda表达式，其目的是使Java更易于为多核处理器编写代码；其次，新加入的Nashorn引擎也使得Java程序可以和JavaScript代码互操作；再者，新的日期时间API、GC改进、并发改进也相当令人期待。

经过2年半的努力、屡次的延期和9个里程碑版本，甲骨文的Java开发团队终于发布了Java 8正式版本。

Java 8版本最大的改进就是Lambda表达式，其目的是使Java更易于为多核处理器编写代码；其次，新加入的Nashorn引擎也使得Java程序可以和JavaScript代码互操作；再者，新的日期时间API、GC改进、并发改进也相当令人期待。

另外，原本要加入Java 8的Jigsaw项目（标准模块系统）由于开发时间关系，被推迟到了Java 9中，不过Java 8已经在朝着这个方向努力了。

Java 8的所有新特性及改进包括(JEP全称为JDK Enhancement Proposal，JDK改进建议)

语言改进：

JEP 126：Lambda表达式 & 虚拟扩展方法

JEP 138：基于Autoconf的构建系统

JEP 160：针对Method Handles的Lambda形式的表征

JEP 161：简洁的配置文件

JEP 162：为模块化做准备

JEP 164：利用CPU指令改善AES加密的性能

JEP 174：Nashorn引擎，允许在Java程序中嵌入JS代码

JEP 176：自动检测识别 Caller-Sensitive 方法

JEP 179：JDK API变化和稳定性记录

VM基础改进：

JEP 142：减少指定字段上的缓存争用

VM垃圾回收（vm/gc）改进：

JEP 122：移除Permanent Generation（永久代）

JEP 173：移除一些少使用的垃圾回收器组合

VM运行时（vm/rt）改进：

JEP 136：提供更多的验证错误信息

JEP 147：减少类元数据封装

JEP 148：支持创建小型虚拟机（3M以下）

JEP 171：添加3个内存有序化的内联函数

核心基础（core）改进：

JEP 153：命令行启动JavaFX应用

核心lang（core/lang）改进：

JEP 101：目标类型推断

JEP 104：Java类型注解

JEP 105：DocTree API

JEP 106：在javax.tools中添加Javadoc

JEP 117：移除APT（Annotation-Processing Tool）

JEP 118：运行过程中可访问参数名

JEP 120：重复注解

JEP 139：增强了javac，以改善构建速度

JEP 172：DocLint工具，用来检查Javadoc注释内容

核心库（core/libs）改进：

JEP 103：并行数组排序

JEP 107：集合数据批量操作

JEP 109：增强的包含Lambda的核心库

JEP 112：改进了字符集的实现

JEP 119：Core Reflection提供的javax.lang.model实现

JEP 135：Base64编解码

JEP 149：减少了核心库的内存占用

JEP 150：日期时间API

JEP 155：改进对并发的支持

JEP 170：JDBC 4.2

JEP 177：java.text.DecimalFormat.format优化

JEP 178：静态链接的JNI库

JEP 180：使用平衡树处理频繁的HashMap碰撞

核心i18n（core/i18n）改进：

JEP 127：改进了本地数据封装，采用Unicode CLDR数据

JEP 128：BCP 47局部匹配

JEP 133：Unicode 6.2

核心net（core/net）改进：

JEP 184：HTTP URL访问权限

核心安全（core/sec）改进：

JEP 113：MS-SFU Kerberos 5扩展

JEP 114：TLS Server Name Indication（SNI）扩展

JEP 115：AEAD密码套件

JEP 121：更强的口令加密系统算法

JEP 123：可配置的安全随机数生成方法

JEP 124：增强了证书撤回检测API

JEP 129：NSA Suite B加密算法实现

JEP 130：SHA-224消息摘要算法实现

JEP 131：针对64位Windows的SunPKCS11加密提供程序

JEP 140：特权限制

JEP 166：彻底检修JKS-JCEKS-PKCS12密钥库

web/jaxp改进：

JEP 185：JAXP 1.5（限制获取外部资源）

详细信息：http://openjdk.java.net/projects/jdk8/features

JDK 8下载：https://jdk8.java.net/

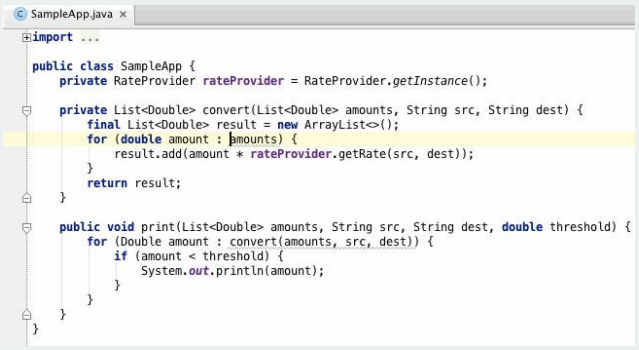
同时发布的还有NetBeans IDE 8.0正式版本，新版本特性见：NetBeans IDE 8.0 新特性一览■

IntelliJ IDEA 13.1 RC2 完成Java 8的最终支持

我总是很希望向大家介绍关于 IntelliJ IDEA 的一些新特性，而这次是对 Java 8 的最终的支持。

IntelliJ IDEA 13.1 RC2 发布了，完成了最终对 Java 8 的支持改进。你应该还记得我专门写了博客介绍支持 Java 8 的代码助手 (in v12 and v13)．现在让我来解释一下v13.1所提供的支持。

除了整体的增强支持外，IntelliJ IDEA 13.1 包含一个全新检视器来帮助你移植代码到心的 Stream API，下面是简单的例子：



你能看到，IDE 同样可以帮你理解 lambda 表达式用的哪个方法，会在相应位置显示可点击的图标。

当你希望使用相应的 quick-fix 在超过一处的代码时，你可以在整个项目中使用这个检视功能，透过：Analyze → Run inspection by Name.

我们希望这个代码辅助可以帮你提升代码并尽快移植到新的 Java 版本。

下载地址：http://www.jetbrains.org/display/IJOS/Download

Java 8那些被冷落的新特性

□ 花名（有孚）

lambda表达式，lambda表达式，还是lambda表达式。一提到Java 8就只能听到这个，但这不过是其中的一个新功能而已，Java 8还有许多新的特性——有一些功能强大的新类或者新的用法，还有一些功能则是早就应该加到Java里了。

这里我准备介绍它的10个我个人认为非常值得了解的新特性。总会有一款适合你的，开始来看下吧。

default方法

这是Java语言的一个新特性，现在接口类里可以包含方法体（这就是default方法）了。这些方法会隐式的添加到实现这个接口的每个子类中。

这使得你可以在不破坏代码的前提下扩展原有库的功能。它绝对是个利器。但从另一个方面来说，这使得接口作为协议，类作为具体实现的界限开始变得有点模糊。但好处就是，它通过一个很优雅的方式使得接口变得更智能，同时还避免了代码冗余，并且扩展类库。不好的地方就是，我估计很快就会看到有在接口方法里获取this引用然后强制转化成某个具体类型的写法了。

终止进程

一旦启动外部进程的话，当这个进程崩溃，挂起，或者CPU到达100%的时候，你就得回来擦屁股了。Process类现在增加了两个新的方法，可以来教训下那些不听话的进程了。

第一个是isAlive()方法，有了它你可以判断进程是否还活着。第二个方法则更加强大，它叫destroyForcibly()，你可以用它来强制的杀掉一个已经超时或者不再需要的进程。

StampedLock

提到这个不禁有点小激动。没有人会喜欢在代码中使用同步。用了它肯定会降低程序的吞吐量，更糟糕的话还会导致进程挂起。尽管这

样，有时候你却不得不选择它。

当多个进程访问一个资源的时候，有多种方法可以进行同步。其中用得最多的一种是ReadWriteLock以及基于它的几种实现。它通过阻塞写线程的方式来允许多个线程并发的读，这样减少了线程之间的竞争。听起来还不错，但实际上这个锁实在是太太太慢了，尤其是当有许多写线程的时候。

因此Java 8引入了一个新的读写锁，叫做StampedLock。它不仅更快，同时还提供了一系列强大的API来实现乐观锁，这样如果没有写操作在访问临界区域的话，你只需很低的开销就能获取到一个读锁。访问结束后你可以查询锁来判断这期间是否发生了写操作，如果有的话再选择进行重试，升级锁，或者放弃这个操作。

这的确是一个非常强大的工具，它本身就值得专门花一篇文章来介绍。这个新玩意儿让我感到非常激动和兴奋，它真的是太棒了。

想了解更多请点击这里。

并发计数器

这是多线程程序会用到的另一个小工具。它提供了简单高效的新接口来实现多线程的并发读写计数器的功能，和AtomicInteger比起来，它要更快一些。相当赞的工具。

Optional

不好，又有空指针了，这是所有Java开发人员的痛处。这估计是有史以来最常见的异常了，至少是1965年以来。

Java 8借鉴了Scala和Haskell，提供了一个新的Optional模板，可以用它来封装可能为空的引用。这绝不是终结空指针的银弹，更多只是使API的设计者可以在代码层面声明一个方法可能会返回空值，调用方应该注意这种情况。正因为这个，这只对新的API有效，前提是调用方不要让引用逃逸出封装类，否则的话引用可能会在外面被不安全的废弃掉。

我对这个新的特性真的是又爱又恨。一方面，空指针是一个大问题，只要能解决这个问题的东西我都欢迎。但另一方面，我对它是否能担此重任执怀疑的态度。这是由于使用它的话需要全公司的集体努力，短期内很难会有见效。除非大力地推广，否则很可能会功亏一篑。

万物皆可注解

还有一个小的改进就是现在Java注解可以支持任意类型了。之前只有像类和方法声明之类的才能使用注解。在Java 8里面，当类型转化甚至分配新对象的时候，都可以在声明变量或者参数的时候使用注解。这是Java为了更好

地支持静态分析及检测工具（比如FireBug)而做的工作中的一部分。这是个很不错的特性，但是和Java 7的invokeDynamic一样，它的真正价值取决于社区以后如何去使用它。

数值溢出

这些方法早就该出现在Java的核心类库里了。我有个癖好就是去测试整型超出2^32时溢出的情况，搞出一些恶心的随机BUG来（怎么会得到这么奇怪的一个值？）。

同样的，这也不是什么银弹，只不过是提供了一组函数，这样你在使用+/*操作符进行数值操作的时候，如果出现了溢出，会抛一个异常。如果我可以决定的话，我会把它作为JVM的默认模式，显式的标明函数会出现数值溢出。

目录遍历

遍历目录树这种事通常都得上Google搜下怎么实现（你很可能用的是Apache.FileUtils）。Java 8给Files类做了一次整容手术，增加了十个新的方法。我最喜欢的一个是walk()方法，它遍历目录后会创建出一个惰性的流（文件系统很大的情况下非常有用）。

增强的随机数生成

现在经常都在讨论密码或者密钥容易遭受攻击的事。程序的安全性是项很复杂的工程，并且很容易出错。这就是我为什么喜欢这个新的SecureRandom.getinstanceStrong()方法的原因，它能自动选择出当前JVM可用的最佳的随机数生成器。这样减少了获取失败的机率，同时也避免了默认的弱随机数生成器可能会导致密钥或者加密值容易被黑客攻破的问题。

Date.toInstant()

Java 8引入了一个新的日期API。这不难理解，因为现有的这个实在是太难用了。实际上Joda一直以来都是Java日期API的首选。不过尽管有了新的API，但仍有一个严重的问题——大量的旧代码和库仍然在使用老的API。

并且我们还知道这种现状仍将继续存在下去。到底该怎么做呢？

Java 8很优雅的解决了这个问题，它给Date类增加了一个新的方法toInstant()，它可以将Date转化成新的实现。这样你马上就可以切换到新的API，尽管现有的代码还在使用老的日期API（并且在可预见的未来仍将继续这样）。

如果你觉得有什么遗漏的或者你觉得我有什么讲的不对的地方，请不吝赐教。下面的评论框就是为这个而准备的:-)■

Java7与Java8JSR核准通过

本文来自：开源中国社区

最近Java JSR经核准通过，但Apache全部投了反对票。Google与Tim Peierls则对Java SE 7与Java SE 8 JSR投了反对票，以此在闹得沸沸扬扬的TCK许可与使用限制这个问题上发出了自己的声音。

· Project Coin，JSR 334，13票通过，1票反对，1票弃权

· Project Lambda，JSR 335，13票通过，1票反对，1票弃权

· Java SE 7，JSR 336，12票通过，3票反对

· Java SE 8，JSR 337，12票通过，3票反对

相关的评论很有趣：Steven Colebourne在一个网页上总结了相关公司的评论。虽说大多数都对TCK投了赞成票，但相关的评论却对其许可问题提出了批评：

· **Google：**投了反对票，因为其许可条款

· **SAP AG：**虽然我们相信Java 7的继续发展很重要，但我们想对Oracle就Apache TCK的决定表达自己的不同观点

· **Eclipse：**对围绕着Java许可的纷纷扰扰感到非常失望

· **RedHat：**对许可条款感到极度失望，规范领导并没有采取更加开放的许可

· **Credit Suisse：**目前，围绕着许可条款的明争暗斗揭示出Java始终没有形成一个开放的标准

众多的评论还表明只要JSR能让Java平台从现在停滞的状态向前迈进，那么这些公司就会投JSR的赞成票。此外，模块化在Java SE 7与Java SE 8的讨论中也被多次提到，Java SE 8 JSR还特别提到了OSGi的互操作性。

这也是各大公司首次通过投票表决的方式来核准Java SE JSR。同时，最近Project Coin与Project Lambda上的工作取得了很大的进展，Java SE 7中的其他内容（比如JDBC 4.1）也深受人们欢迎，Java SE 8 JSR还包含了大量处于襁褓中的JSR。或许当Java SE 8发布时，Oracle会说这是大多数人的意愿，即便Java SE 8的内容与之前的版本有很大差别。

然而，由于许可问题并未得到解决，因此一些人称JCP“仅仅是一些客户而已”——Oracle不再认真对待JCP了。

这场争斗最后的结果就是Apache宣布离开JCP，将继续追寻自己的脚步。这也许是Apache

软件基金会最后一次对JSR投票了：

> Apache软件基金会必须对JSR投反对票。虽然我们支持JSR的技术内容，也认为Java平台需要向前发展，但凭良心说，我们得对这个JSR投反对票，原因在于：
>
> 1.该JSR的TCK许可包含了一个“使用限制”，限制了独立实现的正常使用，这个许可元素不仅被JSPA所禁止，还被 JCP EC的大多数成员所拒绝——包括Oracle。我们只能推测Oracle包含这一限制的原因，但我们认为开放的规范生态系统必须要独立于任何组织的商业利 益。
>
> 2.该JSR与自己的TCK许可自相矛盾。JSR显式声明Java SE以嵌入式部署作为目标。但TCK许可则特别明确地禁止了经过测试的独立实现的使用（比如说上网本）。我们认为这会对潜在的实现者造成误导，通过TCK 的任何独立实现都应该可以使用并且根据实现者所提供的条款进行分发。
>
> 3.规范领导忽视了多个EC成员的再三请求
>
> 4.规范领导——Oracle——违背了身处JSPA的义务——为Apache Harmony提供TCK许可，让Apache根据自己的选择分发其独立实现。我们认为故意不履行JSPA义务的任何人都没资格成为JCP的成员。这个原则适用于所有人。
>
> 虽然我们理解Oracle最初的意图是不管EC的决定是什么都要推进Java不断前进，但我们还是奉劝Oracle尽快解决上面提到的那些问题，然后在JCP的体系下继续与JCP成员并肩作战让Java活力重现。
>
> 由于Oracle并不遵守协议，因此Apache软件基金会也发表了自己的声明。■

Java SE 8 在并发工具方面的加强

新的类以及接口

java.util.concurrent 中增加了两个接口四个类：

1．CompletableFuture.AsynchronousCompletionTask接口：标识在async方法中执行的异步任务。

2.CompletionStage<T>接口：异步计算中可能出现的一个阶段，也就是说当一个CompletionStage 完成时执行的动作或计算。

3.CompletableFuture<T>类：一个可以确定完成状态的Future。

4.ConcurrentHashMap.KeySetView<K,V>类：ConcurrentHashMap 的键的集合视图。

5.CountedCompleter<T>类：一个在没有其他action等待的情况下，会执行一个完成action的ForkJoinTask 。

6.CompletionException类：异常类。

ConcurrentHashMap增加新方法

在Java 8中，集合框架基于streams和Lambda表达式做了全新调整：

ConcurrentHashMap增加了30多个方法，包括foreach系列(forEach,forEachKey,forEachValue, forEachEntry)、search系列(search, searchKeys, searchValues, searchEntries)、reduce系列(reduce, reduceToDouble, reduceToLong)以及mappingCount 、newKeySet等方法， 增强后的ConcurrentHashMap更适合做缓存了。

详细：Java SE 8 在并发工具方面的加强

原文来自：成熟的毛毛虫博客

Java 8新特性探究 前言

网名：成熟的毛毛虫，恒拓开源 架构师

自2013年6月13日，oracle就已经发布的Java 8特性完备版本（M7），但最终GA版本将在2014年3月18日（已第二次跳票，原计划今年9月发布的，据官网宣称是为了解决安全问题.....），相信大家多多少少都听闻了关于Java 8的改进，总得来说，Java 8从语言，核心库，国际化、虚拟机，安全性，平台等方面一共有55个特性，本系列博文将带大家对这55个特性进行探究，不求做到最好，只求详细，深入浅出，通俗易懂。

java几个重大版本

java从1995年发布到现在，也走过18年了，个人认为，其中几个java版本都肩负着重大使命，影响甚远；

jdk1.0 1995年5月23日诞生，Oak语言改名为Java，并提出“Write Once，Run anywhere”；

jdk1.2 1999年6月发布，将java划分为J2SE,J2ME,J2EE三大平台；

jdk1.4 主要是性能提升，在2000年时候JAVA成为世界上最流行的电脑语言，跟这个版本离不开关系，估计国内还有大量的java应用是运行在此版本上；

jdk5 诞生于2004年，他的使命就是易用，加入1.泛型 2 自动装箱/拆箱 3 for-each 4 static import 5 变长参数等， 为了表示该版本的重要性，J2SE1.5更名为Java SE 5.0；

jdk8 将在2014年3月份发布,迄今为止，可能是最大更新的java版本，也是令人期待的一个版本，在Java中引入闭包概念对Java程序开发方法的影响甚至会大于Java5中引入的泛型特征对编程方式带来的影响。

可以看出，jdk 8跟jdk 5之间，整整相差10年，这10年期间，相继发布jdk 6、7都是改动不大，这也说明，java发展确实有点缓慢了，以致曾经的跟随者.net在某方面超越了java，还有基于jvm上的动态语言崛起，比如Groovy、Scala等，2013年java one大会上，java平台首席架构师Mark Reinhold在会上说Java 8 is Revolutionary, Java is back（Java 8是革命性的,Java回来了），至于java 8能否能给我们带来眼前一亮的感觉呢，我们拭目以待吧

学习java8的理由

1.提高java开发效率（更少的代码，更强的功能，主要是lambda表达式带来便利）

2.提高java程序的运行速度（批量数据处理，多核运行利用，更高的性能）

3.更安全，漏洞更少（为啥GA版本跳票，oracle解释说近来大量安全问题困扰着平台，所以推迟发布时间解决安全问题）

4.面试需要，跳槽的哥们，个人预言渐渐会被问及java 8的特性，想当初java 5出来一两年都会

5.逼格高，想在同事面前炫耀一般，但请注意，别以为java是向下兼容的就能随便把旧java系统升级到java最新版本

6.还在靠java开发养家糊口的程序员，不想知识陈旧落后。

Java 8新特性探究（一）：通往lambda之路_语法篇

□ quasimodo_es

现在开始要灌输一些概念性的东西了，这能帮助你理解lambda更加透彻一点，如果你之前听说过，也可当是温习，所谓温故而知新.....

在开始之前，可以同步下载jdk 8 和 IDE，IDE根据个人习惯了，不过eclipse官方版本还没出来，所以目前看的话，netbean7.4是首选的，毕竟前段子刚刚出的正式版本，以下是他们的下载地址。

jdk 8：https://jdk8.java.net/download.html （毕竟是国外的网站，如果下载慢，可以到我的云盘下载http://pan.baidu.com/share/link?shareid=61064476&uk=4060588963）

Netbeans 7.4正式版:https://netbeans.org/downloads/(推荐，oracle官方发布)

IDEA 12 EAP http://confluence.jetbrains.net/display/IDEADEV/IDEA+12+EAP

Unofficial builds of Eclipse :http://downloads.efxclipse.org/eclipse-java8/

函数式接口

函数式接口（functional interface 也叫功能性接口，其实是同一个东西）。简单来说，函数式接口是只包含一个方法的接口。比如Java标准库中的java.lang.Runnable和 java.util.Comparator都是典型的函数式接口。java 8提供 @FunctionalInterface作为注解,这个注解是非必须的，只要接口符合函数式接口的标准（即只包含一个方法的接口），虚拟机会自动判断，但最好在接口上使用注解@FunctionalInterface进行声明，以免团队的其他人员错误地往接口中添加新的方法。

Java中的lambda无法单独出现，它需要一个函数式接口来盛放，lambda表达式方法体其实就是函数接口的实现，下面讲到语法会讲到

Lambda语法

包含三个部分

一个括号内用逗号分隔的形式参数，参数是函数式接口里面方法的参数

一个箭头符号：->

方法体，可以是表达式和代码块，方法体函数式接口里面方法的实现，如果是代码块，则必须用{}来包裹起来，且需要一个return 返回值，但有个例外，若函数式接口里面方法返回值是void，则无需{}

总体看起来像这样

```
(parameters) -> expression 或者 (parameters) -> { statements; }
```

看一个完整的例子，方便理解

```
/**
 * 测试lambda表达式
 *
 * @author benhail
 */
public class TestLambda {

    public static void runThreadUseLambda() {
        //Runnable是一个函数接口，只包含了有个无参数的，返回void的run方法；
        //所以lambda表达式左边没有参数，右边也没有return，只是单纯的打印一句话
        new Thread(() ->System.out.println("lambda实现的线程")).start();
    }

    public static void runThreadUseInnerClass() {
        //这种方式就不多讲了，以前旧版本比较常见的做法
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("内部类实现的线程");
            }
        }).start();
    }

    public static void main(String[] args) {
        TestLambda.runThreadUseLambda();
        TestLambda.runThreadUseInnerClass();
    }
}
```

可以看出，使用lambda表达式设计的代码会更加简洁，而且还可读。

方法引用

其实是lambda表达式的一个简化写法，所引用的方法其实是lambda表达式的方法体实现，语法也很简单，左边是容器（可以是类名，实例名），中间是"::"，右边是相应的方法名。如下所示：

```
ObjectReference::methodName
```

一般方法的引用格式是

1、如果是静态方法，则是ClassName::methodName。如 Object ::equals

2、如果是实例方法，则是Instance::methodName。如Object obj=new Object();obj::equals;

3、构造函数.则是ClassName::new

再来看一个完整的例子，方便理解

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
```

```
import javax.swing.JButton;
import javax.swing.JFrame;

/**
 *
 * @author benhail
 */
public class TestMethodReference {

    public static void main(String[] args) {

        JFrame frame = new JFrame();
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);

        JButton button1 = new JButton("点我!");
        JButton button2 = new JButton("也点我!");

        frame.getContentPane().add(button1);
        frame.getContentPane().add(button2);
        //这里addActionListener方法的参数是ActionListener，是一个函数式接口
        //使用lambda表达式方式
        button1.addActionListener(e -> { System.out.println("这里是Lambda实现方式"); });
        //使用方法引用方式
        button2.addActionListener(TestMethodReference::doSomething);
    }

    /**
     * 这里是函数式接口ActionListener的实现方法
     * @param e
     */
    public static void doSomething(ActionEvent e) {

        System.out.println("这里是方法引用实现方式");
    }
}
```

可以看出，doSomething方法就是lambda表达式的实现，这样的好处就是，如果你觉得lambda的方法体会很长，影响代码可读性，方法引用就是个解决办法

总结

以上就是lambda表达式语法的全部内容了，相信大家对lambda表达式都有一定的理解了，但只是代码简洁了这个好处的话，并不能打动很多观众，java 8也不会这么令人期待，其实java 8引入lambda迫切需求是因为lambda 表达式能简化集合上数据的多线程或者多核的处理，提供更快的集合处理速度 ，这个后续会讲到，关于JEP126的这一特性，将分3部分，之所以分开，是因为这一特性可写的东西太多了，这部分让读者熟悉lambda表达式以及方法引用的语法和概念，第二部分则是虚拟扩展方法（default method）的内容，最后一部分则是大数据集合的处理，解开lambda表达式的最强作用的神秘面纱。敬请期待。■

上篇讲了lambda表达式的语法,但只是JEP126特性的一部分,另一部分就是默认方法（也称为虚拟扩展方法或防护方法）

Java 8新特性探究（二）深入解析默认方法

什么是默认方法，为什么要有默认方法

简单说，就是接口可以有实现方法，而且不需要实现类去实现其方法。只需在方法名前面加个default关键字即可。

为什么要有这个特性？首先，之前的接口是个双刃剑，好处是面向抽象而不是面向具体编程，缺陷是，当需要修改接口时候，需要修改全部实现该接口的类，目前的java 8之前的集合框架没有foreach方法，通常能想到的解决办法是在JDK里给相关的接口添加新的方法及实现。然而，对于已经发布的版本，是没法在给接口添加新方法的同时不影响已有的实现。所以引进的默认方法。他们的目的是为了解决接口的修改与现有的实现不兼容的问题。

简单的例子

一个接口A，Clazz类实现了接口A。

```
public interface A {
    default void foo(){
        System.out.println("Calling A.foo()");
    }
}

public class Clazz implements A {
    public static void main(String[] args){
        Clazz clazz = new Clazz();
        clazz.foo();//调用A.foo()
    }
}
```

代码是可以编译的，即使Clazz类并没有实现foo()方法。在接口A中提供了foo()方法的默认实现。

Java 8抽象类与接口对比

这一个功能特性出来后，很多同学都反应了，java 8的接口都有实现方法了，跟抽象类还有什么区别？其实还是有的，请看下表对比。

相同点

1.都是抽象类型；

2.都可以有实现方法（以前接口不行）；

3.都可以不需要实现类或者继承者去实现所有方法，（以前不行，现在接口中默认方法不需要实现者实现）

不同点

1.抽象类不可以多重继承，接口可以（无论是

多重类型继承还是多重行为继承）；

2.抽象类和接口所反映出的设计理念不同。其实抽象类表示的是”is-a”关系，接口表示的是”like-a”关系；

3.接口中定义的变量默认是public static final 型，且必须给其初值，所以实现类中不能重新定义，也不能改变其值；抽象类中的变量默认是 friendly 型，其值可以在子类中重新定义，也可以重新赋值。

多重继承的冲突说明

由于同一个方法可以从不同接口引入，自然而然的会有冲突的现象，默认方法判断冲突的规则如下：

1.一个声明在类里面的方法优先于任何默认方法（classes always win）

2.否则，则会优先选取最具体的实现，比如下面的例子 B重写了A的hello方法。

```
public interface A {
    default void hello() { System.out.println("Hello World from A"); }
}
public interface B extends A {
    default void hello() { System.out.println("Hello World from B"); }
}
public class C implements B, A {
    public static void main(String... args) {
        new C().hello();
    }
}
```

输出结果是：Hello World from B

如果想调用A的默认函数，则用到新语法X.super.m(...),下面修改C类，实现A接口，重写一个hello方法，如下所示：

```
public class C implements A{
    @Override
    public void hello(){
        A.super.hello();
    }

    public static void main(String[] args){
        new C().hello();
    }
}
```

输出结果是：Hello World from A

总结

默认方法给予我们修改接口而不破坏原来的实现类的结构提供了便利，目前java 8的集合框架已经大量使用了默认方法来改进了，当我们最终开始使用Java 8的lambdas表达式时，提供给我们一个平滑的过渡体验。也许将来我们会在API设计中看到更多的默认方法的应用。

跟上篇博文结合起来，就是JEP126的全部了，后面还有54个特性等着我们去探究，为了让大家比较深刻了解lambda，学以致用，下一篇还是lambda的内容，预告一下下篇的标题：《Java 8特性探究（三）解开lambda表达式最强作用的神秘面纱》，第二个特性 将从第四篇开始，谢谢大家支持，敬请期待。

■

编者按

这篇文章将深入解析Java集合里面的批量数据操作（bulk operation）

Java 8新特性探究（三）解开lambda最强作用的神秘面纱

我们期待了很久lambda为java带来闭包的概念，但是如果我们不在集合中使用它的话，就损失了很大价值。现有接口迁移成为lambda风格的问题已经通过default methods解决了，在这篇文章将深入解析Java集合里面的批量数据操作（bulk operation），解开lambda最强作用的神秘面纱。

1.关于JSR335

JSR是Java Specification Requests的缩写，意思是Java 规范请求,Java 8 版本的主要改进是 Lambda 项目（JSR 335），其目的是使 Java 更易于为多核处理器编写代码。JSR 335=lambda表达式+接口改进（默认方法）+批量数据操作。加上前面两篇，我们已是完整的学习了JSR335的相关内容了。

2.外部VS内部迭代

以前Java集合是不能够表达内部迭代的，而只提供了一种外部迭代的方式，也就是for或者while循环。

上面的例子是我们以前的做法，也就是所谓的外部迭代，循环是固定的顺序循环。在现在多核的时代，如果我们想并行循环，不得不修改以上代码。效率能有多大提升还说定，且会带来一定的风险（线程安全问题等等）。

要描述内部迭代，我们需要用到Lambda这样的类库,下面利用lambda和Collection.forEach重写上面的循环

```
persons.forEach(p->p.setLastName("Doe"));
```

现在是由jdk 库来控制循环了，我们不需要关心last name是怎么被设置到每一个person对象里面去的，库可以根据运行环境来决定怎么做，并行，乱序或者懒加载方式。这就是内部迭代，客户端将行为p.setLastName当做数据传入api里面。

内部迭代其实和集合的批量操作并没有密切的联系，借助它我们感受到语法表达上的变化。真正有意思的和批量操作相关的是新的流（stream）API。新的java.util.stream包已经添加进JDK 8了。

3.Stream API

流（Stream）仅仅代表着数据流，并没有数据结构，所以他遍历完一次之后便再也无法遍历（这

点在编程时候需要注意，不像Collection，遍历多少次里面都还有数据），它的来源可以是Collection、array、io等等。

3.1 中间与终点方法

流作用是提供了一种操作大数据接口，让数据操作更容易和更快。它具有过滤、映射以及减少遍历数等方法，这些方法分两种：中间方法和终端方法，“流”抽象天生就该是持续的，中间方法永远返回的是Stream，因此如果我们要获取最终结果的话，必须使用终点操作才能收集流产生的最终结果。区分这两个方法是看他的返回值，如果是Stream则是中间方法，否则是终点方法。具体请参照Stream的api。

简单介绍下几个中间方法（filter、map）以及终点方法（collect、sum）

3.1.1 Filter

在数据流中实现过滤功能是首先我们可以想到的最自然的操作了。Stream接口暴露了一个filter方法，它可以接受表示操作的Predicate实现来使用定义了过滤条件的lambda表达式。

```
List persons = ...
Stream personsOver18 = persons.stream().
filter(p -> p.getAge() > 18);//过滤18岁以上的人
```

3.1.2 Map

假使我们现在过滤了一些数据，比如转换对象的时候。Map操作允许我们执行一个Function的实现（Function<T,R>的泛型T,R分别表示执行输入和执行结果），它接受入参并返回。首先，让我们来看看怎样以匿名内部类的方式来描述它：

```
Stream adult= persons
            .stream()
            .filter(p -> p.getAge() > 18)
            .map(new Function() {
              @Override
              public Adult apply(Person person) {
                return new Adult(person);//将大于18岁的人转为成年人
              }
            });
```

现在，把上述例子转换成使用lambda表达式的写法：

```
Stream map = persons.stream()
              .filter(p -> p.getAge() > 18)
              .map(person -> new Adult(person));
```

3.1.3 Count

count方法是一个流的终点方法，可使流的结果最终统计，返回int，比如我们计算一下满足18岁的总人数

```
int countOfAdult=persons.stream()
                 .filter(p -> p.getAge() > 18)
                 .map(person -> new Adult(person))
                 .count();
```

3.1.4 Collect

collect方法也是一个流的终点方法，可收集最终的结果

```
List adultList = persons
        .stream()
        .filter(p -> p.getAge() > 18)
        .map(person -> new Adult(person))
        .collect(Collectors.
toCollection(ArrayList::new));
```

篇幅有限，其他的中间方法和终点方法就不一一介绍了，看了上面几个例子，大家明白这两种方法的区别即可，后面可根据需求来决定使用。

3.2顺序流与并行流

每个Stream都有两种模式：顺序执行和并行执行。

顺序流：

```
List <Person> people = list.getStream.
collect(Collectors.toList());
```

并行流：

```
List <Person> people = list.getStream.parallel().
collect(Collectors.toList());
```

顾名思义，当使用顺序方式去遍历时，每个item读完后再读下一个item。而使用并行去遍历时，数组会被分成多个段，其中每一个都在不同的线程中处理，然后将结果一起输出。

3.2.1并行流原理：

```
List originalList = someData;
split1 = originalList(0, mid);//将数据分小部分
split2 = originalList(mid,end);
new Runnable(split1.process());//小部分执行操作
new Runnable(split2.process());
List revisedList = split1 + split2;//将结果合并
```

大家对hadoop有稍微了解就知道，里面的MapReduce 本身就是用于并行处理大数据集的软件框架，其 处理大数据的核心思想就是大而化小，分配到不同机器去运行map，最终通过reduce将所有机器的结果结合起来得到一个最终结果，与MapReduce不同，Stream则是利用多核技术可将大数据通过多核并行处理，而MapReduce则可以分布式的。

3.2.2顺序与并行性能测试对比

如果是多核机器，理论上并行流则会比顺序流快上一倍，下面是测试代码

```
long t0 = System.nanoTime();

//初始化一个范围100万整数流,求能被2整除的数字，toArray（）是终点方法

int a[]=IntStream.range(0, 1_000_000).filter(p ->
p % 2==0).toArray();

long t1 = System.nanoTime();

//和上面功能一样，这里是用并行流来计算

int b[]=IntStream.range(0, 1_000_000).parallel().
```

```
filter(p -> p % 2==0).toArray();

long t2 = System.nanoTime();

//我本机的结果是serial: 0.06s, parallel 0.02s，证明并行流确实比顺序流快

System.out.printf("serial: %.2fs, parallel %.2fs%n", (t1 - t0) * 1e-9, (t2 - t1) * 1e-9);
```

3.3关于Folk/Join框架

应用硬件的并行性在java 7就有了，那就是java.util.concurrent 包的新增功能之一是一个fork-join 风格的并行分解框架，同样也很强大高效，有兴趣的同学去研究，这里不详谈了，相比Stream.parallel()这种方式，我更倾向于后者。

4.总结

如果没有lambda，Stream用起来相当别扭，他会产生大量的匿名内部类，比如上面的3.1.2map例子，如果没有default method，集合框架更改势必会引起大量的改动，所以lambda+default method使得jdk库更加强大，以及灵活，Stream以及集合框架的改进便是最好的证明。

java 8特性探究系列写了3篇了，作为大餐，将java 8的重量级特性lambda与default method写在前面，下篇上个小菜，荤素搭配，也是语言相关的，JEP104 Java 类型的注解的探究，同时谢谢大家的支持，欢迎提出建议。如果你想了解哪些特性，欢迎给我发留言。 ■

Java 8新特性探究（四）类型注解 复杂还是便捷

本文将介绍java 8的第二个特性：类型注解。

注解大家都知道，从java5开始加入这一特性，发展到现在已然是遍地开花，在很多框架中得到了广泛的使用，用来简化程序中的配置。那充满争议的类型注解究竟是什么？复杂还是便捷?

什么是类型注解

在java 8之前，注解只能是在声明的地方所使用，比如类，方法，属性；java 8里面，注解可以应用在任何地方，比如：

- 创建类实例

new @Interned MyObject();

- 类型映射

myString = (@NonNull String) str;

- implements 语句中

class UnmodifiableList<T> implements @Readonly List<@Readonly T> { ... }

- throw exception声明

void monitorTemperature() throws @Critical TemperatureException { ... }

需要注意的是，类型注解只是语法而不是语义，并不会影响java的编译时间，加载时间，以及运行时间，也就是说，编译成class文件的时候并不包含类型注解。

类型注解的作用

先看看下面代码

```
Collections.emptyList().add("One");
int i=Integer.parseInt("hello");
System.console().readLine();
```

上面的代码编译是通过的，但运行是会分别报UnsupportedOperationException； NumberFormatException；NullPointerException异常，这些都是runtime error；

类型注解被用来支持在Java的程序中做强类型检查。配合插件式的check framework，可以在编译的时候检测出runtime error，以提高代码质量。这就是类型注解的作用了。

check framework

check framework是第三方工具，配合Java的类型注解效果就是1+1>2。它可以嵌入到javac编译器里面，可以配合ant和maven使用，也可以作为eclipse插件。地址是http://types.cs.

washington.edu/checker-framework/。

check framework可以找到类型注解出现的地方并检查，举个简单的例子：

```
import checkers.nullness.quals.*;
public class GetStarted {
  void sample() {
    @NonNull Object ref = new Object();
  }
}
```

使用javac编译上面的类

```
javac -processor checkers.nullness.
NullnessChecker GetStarted.java
```

编译是通过，但如果修改成

```
@NonNull Object ref = null;
```

再次编译，则出现

```
GetStarted.java:5: incompatible types.
found   : @Nullable <nulltype>
required: @NonNull Object
    @NonNull Object ref = null;
                          ^
1 error
```

如果你不想使用类型注解检测出来错误，则不需要processor，直接javac GetStarted.java是可以编译通过的，这是在java 8 with Type Annotation Support版本里面可以，但java 5,6,7版本都不行，因为javac编译器不知道@NonNull是什么东西，但check framework 有个向下兼容的解决方案，就是将类型注解nonnull用/**/注释起来，比如上面例子修改为：

```
import checkers.nullness.quals.*;
public class GetStarted {
  void sample() {
    /*@NonNull*/ Object ref = null;
  }
}
```

这样javac编译器就会忽略掉注释块，但用check framework里面的javac编译器同样能够检测出nonnull错误。

通过类型注解+check framework我们可以看到，现在runtime error可以在编译时候就能找到。

关于JSR 308

JSR 308想要解决在Java 1.5注解中出现的两个问题：

- 在句法上对注解的限制：只能把注解写在声明的地方
- 类型系统在语义上的限制：类型系统还做不到预防所有的bug

JSR 308 通过如下方法解决上述两个问题：

- 对Java语言的句法进行扩充，允许注解出现在更多的位置上。包括：方法接收器（method receivers，译注：例public int size() @Readonly { … }），泛型参数，数组，类型转换，类型测试，

对象创建，类型参数绑定，类继承和throws子句。其实就是类型注解，现在是java 8的一个特性

· 通过引入可插拔的类型系统（pluggable type systems）能够创建功能更强大的注解处理器。类型检查器对带有类型限定注解的源码进行分析，一旦发现不匹配等错误之处就会产生警告信息。其实就是check framework

对JSR308，有人反对，觉得更复杂更静态了，比如 @NotEmpty List<@NonNull String> strings = new ArrayList<@NonNull String>()>

换成动态语言为

var strings = [“one”，“two”];

有人赞成，说到底，代码才是“最根本”的文档。代码中包含的注解清楚表明了代码编写者的意图。当没有及时更新或者有遗漏的时候，恰恰是注解中包含的意图信息，最容易在其他文档中被丢失。而且将运行时的错误转到编译阶段，不但可以加速开发进程，还可以节省测试时检查bug的时间。

总结

并不是人人都喜欢这个特性，特别是动态语言比较流行的今天，所幸，java 8并不强求大家使用这个特性，反对的人可以不使用这一特性，而对代码质量有些要求比较高的人或公司可以采用JSR 308，毕竟代码才是“最基本”的文档，这句话我是赞同的。虽然代码会增多，但可以使你的代码更具有表达意义。对这个特性有何看法，大家各抒己见。■

Java 8新特性探究（五）重复注解（repeating annotations）

JEP120没有太多内容，是一个小特性，仅仅是为了提高代码可读性。这次java 8对注解做了2个方面的改进（JEP 104,JEP120），相信注解会比以前使用得更加频繁了。

知识回顾

前面介绍了：

lambda表达式和默认方法 （JEP 126）

批量数据操作（JEP 107）

类型注解（JEP 104）

注：JEP=JDK Enhancement-Proposal (JDK 增强建议)，每个JEP即一个新特性。

在java 8里面，注解一共有2个改进，一个是类型注解，在上篇已经介绍了，本篇将介绍另外一个注解的改进：重复注解（JEP 120）。

什么是重复注解

允许在同一申明类型（类，属性，或方法）的多次使用同一个注解

一个简单的例子

java 8之前也有重复使用注解的解决方案，但可读性不是很好，比如下面的代码：

```
public @interface Authority {
    String role();
}

public @interface Authorities {
   Authority[] value();
}
public class RepeatAnnotationUseOldVersion {

   @Authorities({@Authority(role="Admin"),@
Authority(role="Manager")})
   public void doSomeThing(){
   }
}
```

由另一个注解来存储重复注解，在使用时候，用存储注解Authorities来扩展重复注解，我们再来看看java 8里面的做法：

```
@Repeatable(Authorities.class)
public @interface Authority {
    String role();
}

public @interface Authorities {
   Authority[] value();
}

public class RepeatAnnotationUseNewVersion {
   @Authority(role="Admin")
   @Authority(role="Manager")
   public void doSomeThing(){ }
}
```

不同的地方是，创建重复注解Authority时，加上@Repeatable,指向存储注解Authorities，在使用时候，直接可以重复使用Authority注解。从上面例子看出，java 8里面做法更适合常规的思维，可读性强一点

总结

JEP120没有太多内容，是一个小特性，仅仅是为了提高代码可读性。这次java 8对注解做了2个方面的改进（JEP 104,JEP120），相信注解会比以前使用得更加频繁了。■

Java 8新特性探究（六）泛型的目标类型推断

■ 汇总、透视、提炼、凝炼，对数据处理来说这些词的意思都差不多，R语言提供了很多函数处理这些事情，还有一些软件包也提供了非常方便的数据汇总功能，方法不胜枚举。

简单理解泛型

泛型是Java SE 1.5的新特性，泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。通俗点将就是“类型的变量”。这种类型变量可以用在类、接口和方法的创建中。

理解Java泛型最简单的方法是把它看成一种便捷语法，能节省你某些Java类型转换(casting)上的操作：

```
List<Apple> box = new ArrayList<Apple>();box.
add(new Apple());Apple apple =box.get(0);
```

上面的代码自身已表达的很清楚：box是一个装有Apple对象的List。get方法返回一个Apple对象实例，这个过程不需要进行类型转换。没有泛型，上面的代码需要写成这样：

```
Apple apple = (Apple)box.get(0);
```

泛型的尴尬

泛型的最大优点是提供了程序的类型安全同时可以向后兼容，但也有尴尬的地方，就是每次定义时都要写明泛型的类型，这样显示指定不仅感觉有些冗长，最主要是很多程序员不熟悉泛型，因此很多时候不能够给出正确的类型参数，现在通过编译器自动推断泛型的参数类型，能够减少这样的情况，并提高代码可读性。

java7的泛型类型推断改进

在以前的版本中使用泛型类型，需要在声明并赋值的时候，两侧都加上泛型类型。例如：

```
Map<String, String> myMap = new
HashMap<String, String>();
```

你可能觉得:老子在声明变量的的时候已经指明了参数类型，为毛还要在初始化对象时再指定？幸好，在Java SE 7中，这种方式得以改进，现在你可以使用如下语句进行声明并赋值：

```
Map<String, String> myMap = new
HashMap<>(); //注意后面的”<>”
```

在这条语句中，编译器会根据变量声明时的泛型类型自动推断出实例化HashMap时的泛型类

型。再次提醒一定要注意new HashMap后面的“<>”，只有加上这个“<>”才表示是自动类型推断，否则就是非泛型类型的HashMap，并且在使用编译器编译源代码时会给出一个警告提示。

但是：Java SE 7在创建泛型实例时的类型推断是有限制的：只有构造器的参数化类型在上下文中被显著的声明了，才可以使用类型推断，否则不行。例如：下面的例子在java 7无法正确编译（但现在在java8里面可以编译，因为根据方法参数来自动推断泛型的类型）：

```
List<String> list = new ArrayList<>();
list.add("A");// 由于addAll期望获得Collection<? extends String>类型的参数，因此下面的语句无法通过
list.addAll(new ArrayList<>());
```

Java8的泛型类型推断改进

java8里面泛型的目标类型推断主要2个：

1.支持通过方法上下文推断泛型目标类型

2.支持在方法调用链路当中，泛型类型推断传递到最后一个方法

让我们看看官网的例子

```
class List<E> {
  static <Z> List<Z> nil() { ... };
  static <Z> List<Z> cons(Z head, List<Z> tail) { ... };
  E head() { ... }
}
```

根据JEP101的特性，我们在调用上面方法的时候可以这样写

```
//通过方法赋值的目标参数来自动推断泛型的类型
List<String> l = List.nil();
//而不是显示的指定类型
//List<String> l = List.<String>nil();
//通过前面方法参数类型推断泛型的类型
List.cons(42, List.nil());
//而不是显示的指定类型
//List.cons(42, List.<Integer>nil());
```

总结

以上是JEP101的特性内容了，Java作为静态语言的代表者，可以说类型系统相当丰富。导致类型间互相转换的问题困扰着每个java程序员，通过编译器自动推断类型的东西可以稍微缓解一下类型转换太复杂的问题。 虽然说是小进步，但对于我们天天写代码的程序员，肯定能带来巨大的作用，至少心情更愉悦了~~说不定在java 9里面，我们会得到一个通用的类型var，像js或者scala的一些动态语言那样^_^

■

Java 8新特性探究（七）深入解析日期和时间-JSR310

□ 成熟的毛毛虫

众所周知，日期是商业逻辑计算一个关键的部分，任何企业应用程序都需要处理时间问题。应用程序需要知道当前的时间点和下一个时间点，有时它们还必须计算这两个时间点之间的路径。但java之前的日期做法太令人恶心了，我们先来吐槽一下

吐槽java.util.Date跟Calendar

Tiago Fernandez做过一次投票，选举最烂的JAVA API，排第一的EJB2.X，第二的就是日期API。

槽点一

最开始的时候，Date既要承载日期信息，又要做日期之间的转换，还要做不同日期格式的显示，职责较繁杂（不懂单一职责，你妈妈知道吗？纯属恶搞~哈哈）

后来从JDK 1.1 开始，这三项职责分开了：

・使用Calendar类实现日期和时间字段之间转换；

・使用DateFormat类来格式化和分析日期字符串；

・而Date只用来承载日期和时间信息。

原有Date中的相应方法已废弃。不过，无论是Date，还是Calendar，都用着太不方便了，这是API没有设计好的地方。

槽点二

坑爹的year和month

```
Date date = new Date(2012,1,1);
System.out.println(date);
输出Thu Feb 01 00:00:00 CST 3912
```

观察输出结果，year是2012+1900，而month，月份参数我不是给了1吗？怎么输出二月（Feb）了？

应该曾有人告诉你，如果你要设置日期，应该使用 java.util.Calendar，像这样...

```
Calendar calendar = Calendar.getInstance();
calendar.set(2013, 8, 2);
```

这样写又不对了，calendar的month也是从0开始的，表达8月份应该用7这个数字，要么就干脆用枚举

```
calendar.set(2013, Calendar.AUGUST, 2);
```

注意上面的代码，Calendar年份的传值不需要减去1900（当然月份的定义和Date还是一样），这种不一致真是让人抓狂！

有些人可能知道，Calendar相关的API是IBM捐出去的，所以才导致不一致。

槽点三

java.util.Date与java.util.Calendar中的所有属性都是可变的

下面的代码，计算两个日期之间的天数....

```
public static void main(String[] args) {
    Calendar birth = Calendar.getInstance();
    birth.set(1975, Calendar.MAY, 26);
    Calendar now = Calendar.getInstance();
    System.out.println(daysBetween(birth, now));
    System.out.println(daysBetween(birth, now)); // 显示 0?
 }
public static long daysBetween(Calendar begin, Calendar end) {
    long daysBetween = 0;
    while(begin.before(end)) {
        begin.add(Calendar.DAY_OF_MONTH, 1);
        daysBetween++;
    }
    return daysBetween;
}
```

daysBetween有点问题，如果连续计算两个Date实例的话，第二次会取得0，因为Calendar状态是可变的，考虑到重复计算的场合，最好复制一个新的Calendar

```
public static long daysBetween(Calendar begin, Calendar end) {
    Calendar calendar = (Calendar) begin.clone(); // 复制
    long daysBetween = 0;
    while(calendar.before(end)) {
        calendar.add(Calendar.DAY_OF_MONTH, 1);
        daysBetween++;
    }
    return daysBetween;
}
```

JSR310

以上种种，导致目前有些第三方的java日期库诞生，比如广泛使用的JODA-TIME，还有Date4j等，虽然第三方库已经足够强大，好用，但还是有兼容问题的，比如标准的JSF日期转换器与joda-time API就不兼容，你需要编写自己的转换器，所以标准的API还是必须的，于是就有了JSR310。

JSR 310实际上有两个日期概念。第一个是Instant，它大致对应于java.util.Date类，因为它代表了一个确定的时间点，即相对于标准Java纪元（1970年1月1日）的偏移量；但与java.util.Date类不同的是其精确到了纳秒级别。

第二个对应于人类自身的观念，比如LocalDate和LocalTime。他们代表了一般的时区概念，要么是日期（不包含时间），要么是时间（不包含日期），类似于java.sql的表示方式。此外，还有一个MonthDay，它可以存储某人的

生日（不包含年份）。每个类都在内部存储正确的数据而不是像java.util.Date那样利用午夜12点来区分日期，利用1970-01-01来表示时间。

目前Java8已经实现了JSR310的全部内容。新增了java.time包定义的类表示了日期-时间概念的规则，包括instants，durations，dates，times，time-zones and periods。这些都是基于ISO日历系统，它又是遵循 Gregorian规则的。最重要的一点是值不可变，且线程安全，通过下面一张图，我们快速看下java.time包下的一些主要的类的值的格式，方便理解。

```
• LocalDate        2010-12-03
• LocalTime                11:05:30
• LocalDateTime    2010-12-03T11:05:30
• OffsetTime               11:05:30+01:00
• OffsetDateTime   2010-12-03T11:05:30+01:00
• ZonedDateTime    2010-12-03T11:05:30+01:00 Europe/Paris
• Year             2010
• YearMonth        2010-12
• MonthDay             -12-03
• Instant          2576458258.266 seconds after 1970-01-01
```

方法概览

该包的API提供了大量相关的方法，这些方法一般有一致的方法前缀：

of：静态工厂方法。

parse：静态工厂方法，关注于解析。

get：获取某些东西的值。

is：检查某些东西的是否是true。

with：不可变的setter等价物。

plus：加一些量到某个对象。

minus：从某个对象减去一些量。

to：转换到另一个类型。

at：把这个对象与另一个对象组合起来，例如： date.atTime(time)。

与旧的API对应关系

Java.time ISO Calendar	Java.util Calendar
Instant	Date
LocalDate, LocalTime, LocalDateTime	Calendar
ZonedDateTime	Calendar
OffsetDateTime, OffsetTime,	Calendar
ZoneId, ZoneOffset, ZoneRules	TimeZone
Week Starts on Monday (1 .. 7) enum MONDAY, TUESDAY, ... SUNDAY	Week Starts on Sunday (1 .. 7) int values SUNDAY, MONDAY, ... SATURDAY
12 Months (1 .. 12) enum JANUARY, FEBRUARY, ..., DECEMBER	12 Months (0 .. 11) int values JANUARY, FEBRUARY, ... DECEMBER

简单使用java.time的API

参考http://jinnianshilongnian.iteye.com/blog/1994164 被我揉在一起，可读性很差，相应的代码都有注释了，我就不过多解释了。

```
001  public class TimeIntroduction {
002      public static void testClock() throws InterruptedException {
003          //时钟提供给我们用于访问某个特定 时区的 瞬时时间、日期 和 时间的。
004          Clock c1 = Clock.systemUTC(); //系统默认UTC时钟（当前瞬时时间 System.currentTimeMillis()）
005          System.out.println(c1.millis()); //每次调用将返回当前瞬时时间（UTC）
006          Clock c2 = Clock.systemDefaultZone(); //系统默认时区时钟（当前瞬时时间）
007          Clock c31 = Clock.system(ZoneId.of("Europe/Paris")); //巴黎时区
008          System.out.println(c31.millis()); //每次调用将返回当前瞬时时间（UTC）
009          Clock c32 = Clock.system(ZoneId.of("Asia/Shanghai"));//上海时区
010          System.out.println(c32.millis());//每次调
```

```
用将返回当前瞬时时间（UTC）
011         Clock c4 = Clock.fixed(Instant.now(), ZoneId.of("Asia/Shanghai"));//固定上海时区时钟
012         System.out.println(c4.millis());
013         Thread.sleep(1000);
014         System.out.println(c4.millis()); //不变 即时钟时钟在那一个点不动
015         Clock c5 = Clock.offset(c1, Duration.ofSeconds(2)); //相对于系统默认时钟两秒的时钟
016         System.out.println(c1.millis());
017         System.out.println(c5.millis());
018     }
019     public static void testInstant() {
020         //瞬时时间 相当于以前的System.currentTimeMillis()
021         Instant instant1 = Instant.now();
022         System.out.println(instant1.getEpochSecond());//精确到秒 得到相对于1970-01-01 00:00:00 UTC的一个时间
023         System.out.println(instant1.toEpochMilli()); //精确到毫秒
024         Clock clock1 = Clock.systemUTC(); //获取系统UTC默认时钟
025         Instant instant2 = Instant.now(clock1);//得到时钟的瞬时时间
026         System.out.println(instant2.toEpochMilli());
027         Clock clock2 = Clock.fixed(instant1, ZoneId.systemDefault()); //固定瞬时时间时钟
028         Instant instant3 = Instant.now(clock2);//得到时钟的瞬时时间
029         System.out.println(instant3.toEpochMilli());//equals instant1
030     }
031     public static void testLocalDateTime() {
032         //使用默认时区时钟瞬时时间创建 Clock.systemDefaultZone() -->即相对于 ZoneId.systemDefault()默认时区
033         LocalDateTime now = LocalDateTime.now();
034         System.out.println(now);
035     //自定义时区
036         LocalDateTime now2 = LocalDateTime.now(ZoneId.of("Europe/Paris"));
037         System.out.println(now2);//会以相应的时区显示日期
038     //自定义时钟
039         Clock clock = Clock.system(ZoneId.of("Asia/Dhaka"));
040         LocalDateTime now3 = LocalDateTime.now(clock);
041         System.out.println(now3);//会以相应的时区显示日期
042     //不需要写什么相对时间 如java.util.Date 年是相对于1900 月是从0开始
043     //2013-12-31 23:59
044         LocalDateTime d1 = LocalDateTime.of(2013, 12, 31, 23, 59);
045     //年月日 时分秒 纳秒
046         LocalDateTime d2 = LocalDateTime.of(2013, 12, 31, 23, 59, 59, 11);
047     //使用瞬时时间 + 时区
048         Instant instant = Instant.now();
049         LocalDateTime d3 = LocalDateTime.ofInstant(Instant.now(), ZoneId.systemDefault());
050         System.out.println(d3);
051     //解析String--->LocalDateTime
052         LocalDateTime d4 = LocalDateTime.parse("2013-12-31T23:59");
053         System.out.println(d4);
054         LocalDateTime d5 = LocalDateTime.parse("2013-12-31T23:59:59.999");//999毫秒 等价
```

```
于999000000纳秒
055         System.out.println(d5);
056   //使用DateTimeFormatter API 解析 和 格式化
057         DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy/MM/dd
HH:mm:ss");
058         LocalDateTime d6 = LocalDateTime.
parse("2013/12/31 23:59:59", formatter);
059         System.out.println(formatter.
format(d6));
060   //时间获取
061         System.out.println(d6.getYear());
062         System.out.println(d6.getMonth());
063         System.out.println(d6.getDayOfYear());
064         System.out.println(d6.
getDayOfMonth());
065         System.out.println(d6.getDayOfWeek());
066         System.out.println(d6.getHour());
067         System.out.println(d6.getMinute());
068         System.out.println(d6.getSecond());
069         System.out.println(d6.getNano());
070   //时间增减
071         LocalDateTime d7 = d6.minusDays(1);
072         LocalDateTime d8 = d7.plus(1, IsoFields.
QUARTER_YEARS);
073   //LocalDate 即年月日 无时分秒
074   //LocalTime即时分秒 无年月日
075   //API和LocalDateTime类似就不演示了
076     }
077     public static void testZonedDateTime() {
078         //即带有时区的date-time 存储纳秒、时
区和时差（避免与本地date-time歧义）。
079   //API和LocalDateTime类似，只是多了时差(如
2013-12-20T10:35:50.711+08:00[Asia/Shanghai])
080         ZonedDateTime now = ZonedDateTime.
now();
081         System.out.println(now);
082         ZonedDateTime now2 =
ZonedDateTime.now(ZoneId.of("Europe/Paris"));
083         System.out.println(now2);
084   //其他的用法也是类似的 就不介绍了
085         ZonedDateTime z1 = ZonedDateTime.
parse("2013-12-31T23:59:59Z[Europe/Paris]");
086         System.out.println(z1);
087     }
088     public static void testDuration() {
089         //表示两个瞬时时间的时间段
090         Duration d1 = Duration.between(Instant.
ofEpochMilli(System.currentTimeMillis() -
12323123), Instant.now());
091   //得到相应的时差
092         System.out.println(d1.toDays());
093         System.out.println(d1.toHours());
094         System.out.println(d1.toMinutes());
095         System.out.println(d1.toMillis());
096         System.out.println(d1.toNanos());
097   //1天时差 类似的还有如ofHours()
098         Duration d2 = Duration.ofDays(1);
099         System.out.println(d2.toDays());
100     }
101     public static void testChronology() {
102         //提供对java.util.Calendar的替换，提供
对年历系统的支持
103         Chronology c = HijrahChronology.
INSTANCE;
104         ChronoLocalDateTime d =
c.localDateTime(LocalDateTime.now());
105         System.out.println(d);
106     }
107     /**
```

```
108      * 新旧日期转换
109      */
110     public static void
testNewOldDateConversion(){
111         Instant instant=new Date().toInstant();
112         Date date=Date.from(instant);
113         System.out.println(instant);
114         System.out.println(date);
115     }
116     public static void main(String[] args)
throws InterruptedException {
117         testClock();
118         testInstant();
119         testLocalDateTime();
120         testZonedDateTime();
121         testDuration();
122         testChronology();
123         testNewOldDateConversion();
124     }
125 }
```

与Joda-Time的区别

其实JSR310的规范领导者Stephen Colebourne，同时也是Joda-Time的创建者，JSR310是在Joda-Time的基础上建立的，参考了绝大部分的API，但并不是说JSR310=JODA-Time，下面几个比较明显的区别是

最明显的变化就是包名（从org.joda.time以及java.time）

JSR310不接受NULL值，Joda-Time视NULL值为0

JSR310的计算机相关的时间（Instant）和与人类相关的时间（DateTime）之间的差别变得更明显

JSR310所有抛出的异常都是DateTimeException的子类。虽然DateTimeException是一个RuntimeException

总结

对比旧的日期API

Java.time	java.util.Calendar以及Date
流畅的API	不流畅的API
实例不可变	实例可变
线程安全	非线程安全

日期与时间处理API，在各种语言中，可能都只是个不起眼的API，如果你没有较复杂的时间处理需求，可能只是利用日期与时间处理API取得系统时间，简单做些显示罢了，然而如果认真看待日期与时间，其复杂程度可能会远超过你的想象，天文、地理、历史、政治、文化等因素，都会影响到你对时间的处理。所以在处理时间上，最好选用JSR310（如果你用java8的话就实现310了），或者Joda-Time。

不止是java面临时间处理的尴尬，其他语言同样也遇到过类似的问题，比如

Arrow：Python 中更好的日期与时间处理库

Moment.js：JavaScript 中的日期库

Noda-Time：.NET 阵营的Joda-Time的复制

■

Java 8新特性探究（八）精简的JRE详解

□ 成熟的毛毛虫

Oracle公司如期发布了Java 8正式版！没有让广大javaer失望。对于一个人来说，18岁是人生的转折点，从稚嫩走向成熟，法律意味着你是完全民事行为能力人，不再收益于未成年人保护法，到今年为止，java也走过了18年，java8是一个新的里程碑，带来了前所未有的诸多特性，lambda表达式，Stream API，新的Date time api，多核并发支持，重大安全问题改进等，相信java会越来越好，丰富的类库以及庞大的开源生态环境是其他语言所不具备的，说起丰富的类库，很多同学就吐槽了，java该减肥了，确实是该减肥，java8有个很好的特性，即JEP161(http://openjdk.java.net/jeps/161),该特性定义了Java SE平台规范的一些子集，使java应用程序不需要整个JRE平台即可部署和运行在小型设备上。开发人员可以基于目标硬件的可用资源选择一个合适的JRE运行环境。

好处

1.更小的Java环境需要更少的计算资源。

2.一个较小的运行时环境可以更好的优化性能和启动时间。

3.消除未使用的代码从安全的角度总是好的。

4.这些打包的应用程序可以下载速度更快。

概念

紧凑的JRE分3种，分别是compact1、compact2、compact3，他们的关系是compact1<compact2<compact3,他们包含的API如下图所示

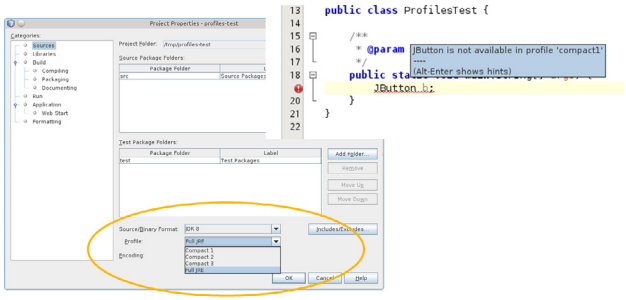
使用javac根据profile编译应用程序

javac - bootclasspath, or javac - profile <profile>

如果不符合compact的api，则报错。

```
$ javac -profile compact2 Test.java
Test.java:7: error: ThreadMXBean is not available in profile 'compact2'
 ThreadMXBean bean = ManagementFactory.getThreadMXBean();
 ^
```

```
Test.java:7: error: ManagementFactory is not
available in profile 'compact2'
 ThreadMXBean bean = ManagementFactory.
getThreadMXBean();
                    ^
2 errors
```



JPEDS工具使用

java8新增一个工具，用来分析应用程序所依赖的profile，有三个参数比较常用 -p，-v，-r

```
import java.util.Set;
import java.util.HashSet;

public class Deps {
  public static void main(String[] args) {
    System.out.println(Math.random());
    Set<String> set = new HashSet<>();
  }
}

************** PROFILE **************
jdeps -P Deps.class
Deps.class -> /Library/Java/
JavaVirtualMachines/jdk1.8.0.jdk/Contents/
Home/jre/lib/rt.jar
   <unnamed> (Deps.class)
      -> java.io
compact1
      -> java.lang
compact1
      -> java.util
compact1

************** VERBOSE
********************
jdeps -v Deps.class
Deps.class -> /Library/Java/
JavaVirtualMachines/jdk1.8.0.jdk/Contents/
Home/jre/lib/rt.jar
   Deps (Deps.class)
      -> java.io.PrintStream
      -> java.lang.Math
      -> java.lang.Object
      -> java.lang.String
      -> java.lang.System
      -> java.util.HashSet

************** RECURSIVE
********************
jdeps -R Deps.class
Deps.class -> /Library/Java/
JavaVirtualMachines/jdk1.8.0.jdk/Contents/
Home/jre/lib/rt.jar
   <unnamed> (Deps.class)
      -> java.io
      -> java.lang
      -> java.util
/Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/
Contents/Home/jre/lib/jce.jar -> /Library/Java/
JavaVirtualMachines/jdk1.8.0.jdk/Contents/
```

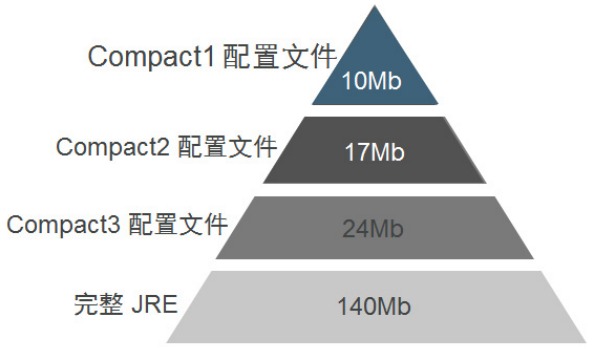
```
Home/jre/lib/rt.jar
  javax.crypto (jce.jar)
    -> java.io
    -> java.lang
    -> java.lang.reflect
    -> java.net
    -> java.nio
    -> java.security
    -> java.security.cert
    -> java.security.spec
    -> java.util
    -> java.util.concurrent
    -> java.util.jar
    -> java.util.regex
    -> java.util.zip
    -> javax.security.auth
    -> sun.security.jca                    JDK
internal API (rt.jar)
    -> sun.security.util                   JDK
internal API (rt.jar)
    -> sun.security.validator
JDK internal API (rt.jar)
  javax.crypto.interfaces (jce.jar)
    -> java.lang
    -> java.math
    -> java.security
  javax.crypto.spec (jce.jar)
    -> java.lang
    -> java.math
    -> java.security.spec
    -> java.util
/Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/
Contents/Home/jre/lib/rt.jar -> /Library/Java/
JavaVirtualMachines/jdk1.8.0.jdk/Contents/
Home/jre/lib/jce.jar
  java.security (rt.jar)
    -> javax.crypto                        JDK
internal API (jce.jar)
  sun.security.util (rt.jar)
    -> javax.crypto                        JDK
internal API (jce.jar)
    -> javax.crypto.interfaces
JDK internal API (jce.jar)
    -> javax.crypto.spec                   JDK
internal API (jce.jar)
```

在linux上构建profile

```
$ hg clone http://hg.openjdk.java.net/jdk8/
jdk8/
$ cd jdk8
$ make images profiles :
## Finished profiles (build time 00:00:27)
----- Build times -------
Start 2013-03-17 14:47:35
End 2013-03-17 14:58:26
00:00:25 corba
00:00:15 demos
00:01:50 hotspot
00:00:24 images
00:00:21 jaxp
00:00:31 jaxws
00:05:37 jdk
00:00:43 langtools
00:00:18 nashorn
00:00:27 profiles
```

```
00:10:51 TOTAL
-------------------------
Finished building Java(TM) for target 'images profiles'
$ cd images
$ ls -d *image
j2re-compact1-image j2re-compact2-image
j2re-compact3-image j2re-image j2sdk-image
```

编译后compact大致的占用空间



总结

如今，物联网正风行一时。我们看到大量不同的设备在市场上出现，每一种的更新速度都越来越快。java需要一个占用资源少的JRE运行环境，紧凑的JRE特性的出现，希望能带来以后的物联网的发展，甚至还是会有大量的java应用程序出现在物联网上面。目前oracle也发布了针对raspberry pi的JRE了。

另外该特性也是为java9的模块化项目做准备，模块化特性是javaer所期待的特性。他是解决业务系统复杂度的一个利器，当然OSGI也是相当的出色。但osgi对于新学者来说未免太复杂了。■

编者按

R语言中进行主成分分析可以采用基本的princomp函数，将结果输入到summary和plot函数中可分别得到分析结果和碎石图。但psych扩展包更具灵活性。

Java 8新特性探究（九）跟OOM：Permgen说再见吧

很多开发者都在其系统中见过“java.lang.OutOfMemoryError: PermGen space”这一问题。这往往是由类加载器相关的内存泄漏以及新类加载器的创建导致的，通常出现于代码热部署时。相对于正式产品，该问题在开发机上出现的频率更高，在产品中最常见的“问题”是默认值太低了。常用的解决方法是将其设置为256MB或更高。

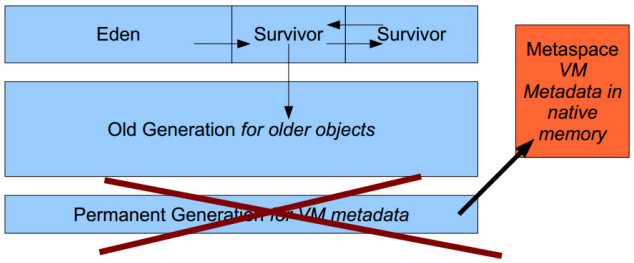
PermGen space简单介绍

PermGen space的全称是Permanent Generation space,是指内存的永久保存区域，说说为什么会内存益出：这一部分用于存放Class和Meta的信息,Class在被 Load的时候被放入PermGen space区域，它和和存放Instance的Heap区域不同,所以如果你的APP会LOAD很多CLASS的话,就很可能出现PermGen space错误。这种错误常见在web服务器对JSP进行pre compile的时候。

JVM 种类有很多，比如 Oralce-Sun Hotspot, Oralce JRockit, IBM J9, Taobao JVM（淘宝好样的！）等等。当然武林盟主是Hotspot了，这个毫无争议。需要注意的是，PermGen space是Oracle-Sun Hotspot才有，JRockit以及J9是没有这个区域。

元空间(MetaSpace)一种新的内存空间诞生

JDK8 HotSpot JVM 将移除永久区，使用本地内存来存储类元数据信息并称之为：元空间（Metaspace）；这与Oracle JRockit 和IBM JVM’s很相似，如下图所示



这意味着不会再有java.lang.OutOfMemoryError: PermGen问题，也不再需要你进行调优及监控内存空间的使用……但请等等，这么说还为时过早。在默认情况下，这些改变是透明的，接下来我们的展示将使你知道仍然要关注类元数据内存的占用。请一定要牢记，这个新特

性也不能神奇地消除类和类加载器导致的内存泄漏。

java8中metaspace总结如下：

PermGen 空间的状况

这部分内存空间将全部移除。

JVM的参数：PermSize 和 MaxPermSize 会被忽略并给出警告（如果在启用时设置了这两个参数）。

Metaspace 内存分配模型

大部分类元数据都在本地内存中分配。

用于描述类元数据的“klasses”已经被移除。

Metaspace 容量

默认情况下，类元数据只受可用的本地内存限制（容量取决于是32位或是64位操作系统的可用虚拟内存大小）。

新参数（MaxMetaspaceSize）用于限制本地内存分配给类元数据的大小。如果没有指定这个参数，元空间会在运行时根据需要动态调整。

Metaspace 垃圾回收

对于僵死的类及类加载器的垃圾回收将在元数据使用达到“MaxMetaspaceSize”参数的设定值时进行。

适时地监控和调整元空间对于减小垃圾回收频率和减少延时是很有必要的。持续的元空间垃圾回收说明，可能存在类、类加载器导致的内存泄漏或是大小设置不合适。

Java 堆内存的影响

一些杂项数据已经移到Java堆空间中。升级到JDK8之后，会发现Java堆 空间有所增长。

Metaspace 监控

元空间的使用情况可以从HotSpot1.8的详细GC日志输出中得到。

Jstat 和 JVisualVM两个工具，在使用b75版本进行测试时，已经更新了，但是还是能看到老的PermGen空间的出现。

前面已经从理论上充分说明，下面让我们通过“泄漏”程序进行新内存空间的观察……

PermGen vs. Metaspace 运行时比较

为了更好地理解Metaspace内存空间的运行时行为，

将进行以下几种场景的测试：

1.使用JDK1.7运行Java程序，监控并耗尽默认设定的85MB大小的PermGen内存空间。

2.使用JDK1.8运行Java程序，监控新Metaspace内存空间的动态增长和垃圾回收过程。

3.使用JDK1.8运行Java程序，模拟耗尽通过“MaxMetaspaceSize”参数设定的128MB大小的Metaspace内存空间。

首先建立了一个模拟PermGen OOM的代码

```
public class ClassA {
 public void method(String name) {
  // do nothing
 }
}
```

上面是一个简单的ClassA，把他编译成class字节码放到D：/classes下面，测试代码中用URLClassLoader来加载此类型上面类编译成class

```
/**
 * 模拟PermGen OOM
 * @author benhail
 */
public class OOMTest {
    public static void main(String[] args) {
        try {
            //准备url
            URL url = new File("D:/classes").toURI().
toURL();
            URL[] urls = {url};
            //获取有关类型加载的JMX接口
            ClassLoadingMXBean loadingBean =
ManagementFactory.
getClassLoadingMXBean();
            //用于缓存类加载器
            List<ClassLoader> classLoaders = new
ArrayList<ClassLoader>();
            while (true) {
                //加载类型并缓存类加载器实例
                ClassLoader classLoader = new
URLClassLoader(urls);
                classLoaders.add(classLoader);
                classLoader.loadClass("ClassA");
                //显示数量信息（共加载过的类型数
目，当前还有效的类型数目，已经被卸载的类
型数目）
                System.out.println("total: " +
loadingBean.getTotalLoadedClassCount());
                System.out.println("active: " +
loadingBean.getLoadedClassCount());
                System.out.println("unloaded: " +
loadingBean.getUnloadedClassCount());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

虚拟机器参数设置如下：-verbose

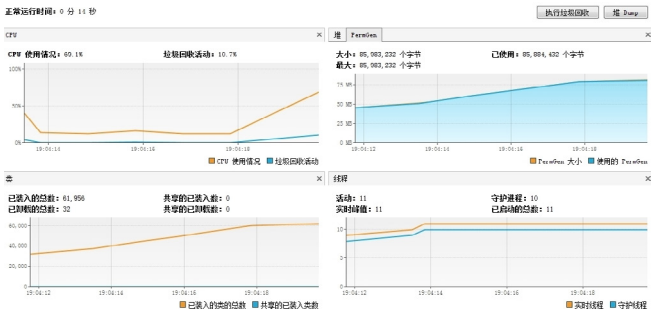
-verbose:gc

设置-verbose参数是为了获取类型加载和卸载的信息

设置-verbose:gc是为了获取垃圾收集的相关信息

JDK 1.7 @64-bit - PermGen 耗尽测试

Java1.7的PermGen默认空间为85 MB（或者可以通过-XX:MaxPermSize=XXXm指定）



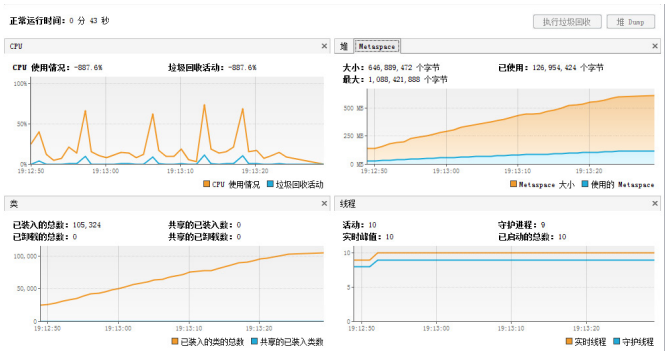
可以从上面的JVisualVM的截图看出：当加载超过6万个类之后，PermGen被耗尽。我们也能通过程序和GC的输出观察耗尽的过程。

程序输出(摘取了部分)

```
.....
[Loaded ClassA from file:/D:/classes/]
total: 64887
active: 64887
unloaded: 0
[GC 245041K->213978K(536768K), 0.0597188 secs]
[Full GC 213978K->211425K(644992K), 0.6456638 secs]
[GC 211425K->211425K(656448K), 0.0086696 secs]
[Full GC 211425K->211411K(731008K), 0.6924754 secs]
[GC 211411K->211411K(726528K), 0.0088992 secs]
.....
java.lang.OutOfMemoryError: PermGen space
```

JDK 1.8 @64-bit - Metaspace大小动态调整测试

Java的Metaspace空间：不受限制 （默认）

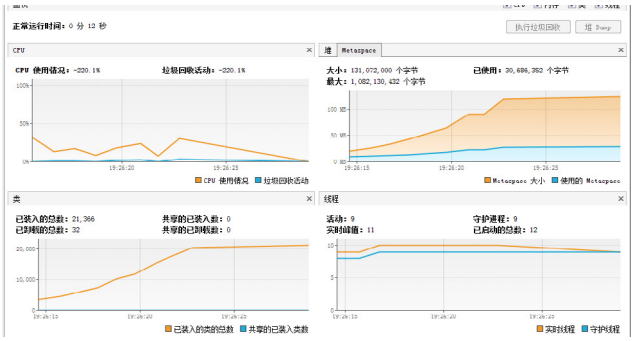


从上面的截图可以看到，JVM Metaspace进行了动态扩展，本地内存的使用由20MB增长到646MB，以满足程序中不断增长的类数据内存占用需求。我们也能观察到JVM的垃圾回收事件—试图销毁僵死的类或类加载器对象。但是，

由于我们程序的泄漏，JVM别无选择只能动态扩展Metaspace内存空间。程序加载超过10万个类，而没有出现OOM事件。

JDK 1.8 @64-bit - Metaspace 受限测试

Java的Metaspace空间：128MB（-XX:MaxMetaspaceSize=128m）



可以从上面的JVisualVM的截图看出：当加载超过2万个类之后，Metaspace被耗尽；与JDK1.7运行时非常相似。我们也能通过程序和GC的输出观察耗尽的过程。另一个有趣的现象是，保留的原生内存占用量是设定的最大大小两倍之多。这可能表明，如果可能的话，可微调元空间容量大小策略，来避免本地内存的浪费。

从Java程序的输出中看到如下异常。

```
[Loaded ClassA from file:/D:/classes/]
total: 21393
active: 21393
unloaded: 0
[GC (Metadata GC Threshold)
64306K->57010K(111616K), 0.0145502 secs]
[Full GC (Metadata GC Threshold)
57010K->56810K(122368K), 0.1068084 secs]
java.lang.OutOfMemoryError: Metaspace
```

在设置了MaxMetaspaceSize的情况下，该空间的内存仍然会耗尽，进而引发“java.lang.OutOfMemoryError: Metadata space”错误。因为类加载器的泄漏仍然存在，而通常Java又不希望无限制地消耗本机内存，因此设置一个类似于MaxPermSize的限制看起来也是合理的。

总结

1.之前不管是不是需要，JVM都会吃掉那块空间……如果设置得太小，JVM会死掉；如果设置得太大，这块内存就被JVM浪费了。理论上说，现在你完全可以不关注这个，因为JVM会在运行时自动调校为“合适的大小”；

2.提高Full GC的性能，在Full GC期间，Metadata到Metadata pointers之间不需要扫描了，别小看这几纳秒时间；

3.隐患就是如果程序存在内存泄露，像OOMTest那样，不停的扩展metaspace的空间，会导致机器的内存不足，所以还是要有必要的调试和监控。■

Java 8 中 HashMap 的性能提升

HashMap是一个高效通用的数据结构，它在每一个Java程序中都随处可见。先来介绍些基础知识。

HashMap是一个高效通用的数据结构，它在每一个Java程序中都随处可见。先来介绍些基础知识。你可能也知道，HashMap使用key的hashCode()和equals()方法来将值划分到不同的桶里。桶的数量通常要比map中的记录的数量要稍大，这样每个桶包括的值会比较少（最好是一个）。当通过key进行查找时，我们可以在常数时间内迅速定位到某个桶（使用hashCode()对桶的数量进行取模）以及要找的对象。

这些东西你应该都已经知道了。你可能还知道哈希碰撞会对hashMap的性能带来灾难性的影响。如果多个hashCode()的值落到同一个桶内的时候，这些值是存储到一个链表中的。最坏的情况下，所有的key都映射到同一个桶中，这样hashmap就退化成了一个链表——查找时间从O(1)到O(n)。我们先来测试下正常情况下hashmap在Java 7和Java 8中的表现。为了能完成控制hashCode()方法的行为，我们定义了如下的一个Key类：

```
class Key implements Comparable<Key> {
private final int value;
Key(int value) {
this.value = value;
}
@Override
public int compareTo(Key o) {
return Integer.compare(this.value, o.value);
}
@Override
public boolean equals(Object o) {
if (this == o) return true;
if (o == null || getClass() != o.getClass())
return false;
Key key = (Key) o;
return value == key.value;
}
@Override
public int hashCode() {
return value;
}
}
```

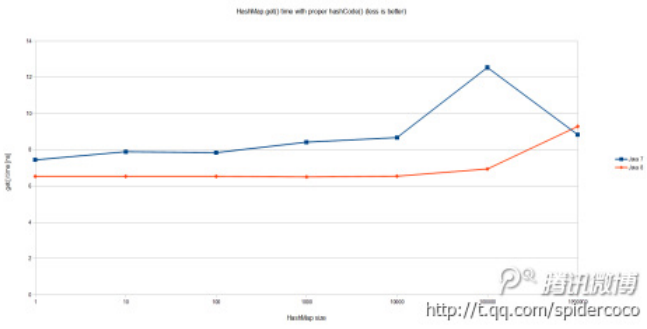
Key类的实现中规中矩：它重写了equals()方法并且提供了一个还算过得去的hashCode()方法。为了避免过度的GC，我将不可变的Key对象缓存了起来，而不是每次都重新开始创建一遍：

```
class Key implements Comparable<Key> {
public class Keys {
```

```
public static final int MAX_KEY = 10_000_000;
private static final Key[] KEYS_CACHE = new
Key[MAX_KEY];
static {
for (int i = 0; i < MAX_KEY; ++i) {
KEYS_CACHE[i] = new Key(i);
}
}
public static Key of(int value) {
return KEYS_CACHE[value];
}
}
```

现在我们可以开始进行测试了。我们的基准测试使用连续的Key值来创建了不同的大小的HashMap（10的乘方，从1到1百万）。在测试中我们还会使用key来进行查找，并测量不同大小的HashMap所花费的时间：

```
import com.google.caliper.Param;
import com.google.caliper.Runner;
import com.google.caliper.SimpleBenchmark;
public class MapBenchmark extends
SimpleBenchmark {
private HashMap<Key, Integer> map;
@Param
private int mapSize;
@Override
protected void setUp() throws Exception {
map = new HashMap<>(mapSize);
for (int i = 0; i < mapSize; ++i) {
map.put(Keys.of(i), i);
}
}
public void timeMapGet(int reps) {
for (int i = 0; i < reps; i++) {
map.get(Keys.of(i % mapSize));
}
}
}
```

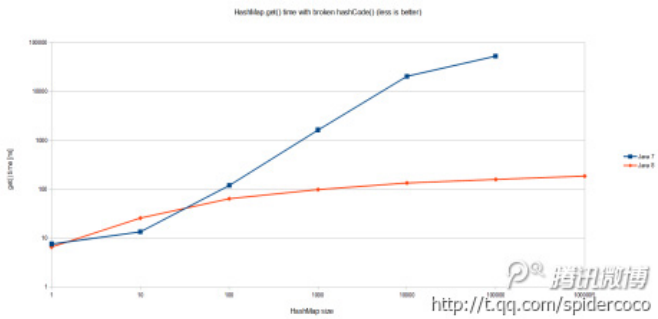


有意思的是这个简单的HashMap.get()里面，Java 8比Java 7要快20%。整体的性能也相当不错：尽管HashMap里有一百万条记录，单个查询也只花了不到10纳秒，也就是大概我机器上的大概20个CPU周期。相当令人震撼！不过这并不是我们想要测量的目标。

假设有一个很差劲的key，他总是返回同一个值。这是最糟糕的场景了，这种情况完全就不应该使用HashMap:

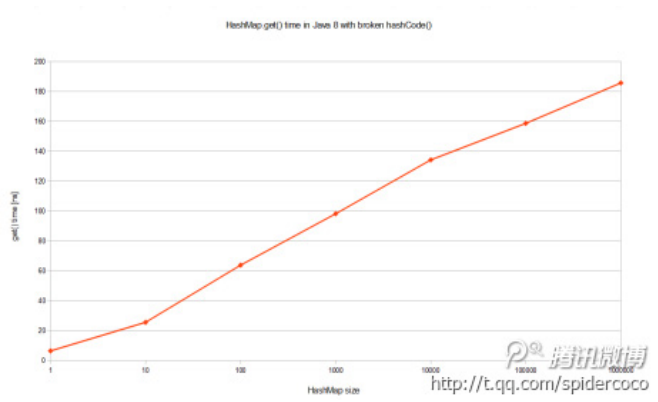
```
class Key implements Comparable<Key> {
//...
@Override
public int hashCode() {
```

```
return 0;
}
}
```



Java 7的结果是预料中的。随着HashMap的大小的增长，get()方法的开销也越来越大。由于所有的记录都在同一个桶里的超长链表内，平均查询一条记录就需要遍历一半的列表。因此从图上可以看到，它的时间复杂度是O(n)。

不过Java 8的表现要好许多！它是一个log的曲线，因此它的性能要好上好几个数量级。尽管有严重的哈希碰撞，已是最坏的情况了，但这个同样的基准测试在JDK8中的时间复杂度是O(logn)。单独来看JDK 8的曲线的话会更清楚，这是一个对数线性分布：



为什么会有这么大的性能提升，尽管这里用的是大O符号（大O描述的是渐近上界）？其实这个优化在JEP-180中已经提到了。如果某个桶中的记录过大的话（当前是TREEIFY_THRESHOLD = 8），HashMap会动态的使用一个专门的treemap实现来替换掉它。这样做的结果会更好，是O(logn)，而不是糟糕的O(n)。它是如何工作的？前面产生冲突的那些KEY对应的记录只是简单的追加到一个链表后面，这些记录只能通过遍历来进行查找。但是超过这个阈值后HashMap开始将列表升级成一个二叉树，使用哈希值作为树的分支变量，如果两个哈希值不等，但指向同一个桶的话，较大的那个会插入到右子树里。如果哈希值相等，HashMap希望key值最好是实现了Comparable接口的，这样它可以按照顺序来进行插入。这对HashMap的key来说并不是必须的，不过如果实现了当然最好。如果没有实现这个接口，在出现严重的哈希碰撞的时候，你就并别指望能获得性能提升了。

这个性能提升有什么用处？比方说恶意的程序，如果它知道我们用的是哈希算法，它可能会发送大量的请求，导致产生严重的哈希碰撞。然后不停的访问这些key就能显著的影响服务器的性能，这样就形成了一次拒绝服务攻击（DoS）。JDK 8中从O(n)到O(logn)的飞跃，可以有效地防止类似的攻击，同时也让HashMap性能的可预测性稍微增强了一些。我希望这个提升能最终说服你的老大同意升级到JDK 8来。

测试使用的环境是：Intel Core i7-3635QM @ 2.4 GHz，8GB内存，SSD硬盘，使用默认的JVM参数，运行在64位的Windows 8.1系统上。■

Java8 如何进行stream reduce,collection操作

R语言提供了一种方法来对一组数据运行常用统计测试(例如线性和非线性建模、时间序列分析、分类和聚类)，通常结果是以图形的形式出现。

一、概念介绍

在java8 JDK包含许多聚合操作（如平均值，总和，最小，最大，和计数），返回一个计算流stream的聚合结果。这些聚合操作被称为聚合操作。JDK除返回单个值的聚合操作外，还有很多聚合操作返回一个collection集合实例。很多的reduce操作执行特定的任务，如求平均值或按类别分组元素。

JDK提供的通用的聚合操作：Stream.reduce,Stream.collection

注意：本文将reduction operations翻译为聚合操作，因为reduction operations通常用于汇聚统计。

两者的区别：

Stream.reduce，常用的方法有average, sum, min, max, and count，返回单个的结果值，并且reduce操作每处理一个元素总是创建一个新值

Stream.collection与stream.reduce方法不同，Stream.collect修改现存的值，而不是每处理一个元素，创建一个新值

二、源代码

```
package lambda;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class LambdaMapReduce {
    private static List<User> users = Arrays.asList(
    new User(1, “张三”, 12,User.Sex.MALE),
    new User(2, “李四”, 21, User.Sex.FEMALE),
    new User(3,”王五”, 32, User.Sex.MALE),
    new User(4, “赵六”, 32, User.Sex.FEMALE));

    public static void main(String[] args) {
        reduceAvg();
        reduceSum();

    //与stream.reduce方法不同，Stream.collect修改现存的值，而不是每处理一个元素，创建一个新值
    //获取所有男性用户的平均年龄
Averager averageCollect = users.parallelStream()
.filter(p -> p.getGender() == User.Sex.MALE)
.map(User::getAge)
.collect(Averager::new, Averager::accept,
Averager::combine);

System.out.println(“Average age of male members: “
```

```
        + averageCollect.average());
//获取年龄大于12的用户列表
List<User> list = users.parallelStream().filter(p ->
p.age > 12)
.collect(Collectors.toList());
System.out.println(list);

//按性别统计用户数
Map<User.Sex, Integer> map = users.
parallelStream().collect(
Collectors.groupingBy(User::getGender,
Collectors.summingInt(p -> 1)));
System.out.println(map);
//按性别获取用户名称
Map<User.Sex, List<String>> map2 = users.stream()
        .collect(
        Collectors.groupingBy(
        User::getGender,
        Collectors.mapping(User::getName,
        Collectors.toList())));
        System.out.println(map2);

        //按性别求年龄的总和
        Map<User.Sex, Integer> map3 = users.
stream().collect(
        Collectors.groupingBy(User::getGender,
        Collectors.reducing(0, User::getAge,
Integer::sum)));
        System.out.println(map3);

        //按性别求年龄的平均值
        Map<User.Sex, Double> map4 = users.
stream().collect(
        Collectors.groupingBy(User::getGender,

Collectors.averagingInt(User::getAge)));
        System.out.println(map4);
        }

// 注意，reduce操作每处理一个元素总是创建一个
新值，
// Stream.reduce适用于返回单个结果值的情况
//获取所有用户的平均年龄
private static void reduceAvg() {
        // mapToInt的pipeline后面可以是
average,max,min,count,sum
        double avg = users.parallelStream().
mapToInt(User::getAge)
        .average().getAsDouble();
        System.out.println("reduceAvg User Age: " +
avg);
        }
        //获取所有用户的年龄总和
        private static void reduceSum() {
                double sum = users.parallelStream().
mapToInt(User::getAge)
        .reduce(0, (x, y) -> x + y); // 可以简写为.sum()
                System.out.println("reduceSum
User Age: " + sum);
        }
}
```

■

Java8 default methods 默认方法的概念与代码解析

cloud-coder /文

摘要 Java 8引入default method，或者叫virtual extension method，目的是为了让接口可以事后添加新方法而无需强迫所有实现该接口的类都提供新方法的实现。也就是说它的主要使用场景可能会涉及代码演进。

一、基本概念

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

默认方法使您能够添加新的功能到你现有库的接口中，并确保与采用老版本接口编写的代码的二进制兼容性。

什么是默认方法(default methods)

即接口可以有实现方法，而且不需要实现类去实现其方法。只需在方法名前面加个default关键字即可，这些方法默认是

为什么要有默认方法

为什么要有这个特性?首先，之前的接口是个双刃剑，好处是面向抽象而不是面向具体编程，缺陷是，当需要修改接口时候，需要修改全部实现该接口的类，目前的java 8之前的集合框架没有foreach方法，通常能想到的解决办法是在JDK里给相关的接口添加新的方法及实现。然而，对于已经发布的版本，是没法在给接口添加新方法的同时不影响已有的实现。所以引进的默认方法。他们的目的是为了解决接口的修改与现有的实现不兼容的问题

二、 java 8抽象类与接口的区别

相同点：

1.都是抽象类型;

2.都可以有实现方法（java8才可以）

3.都可以不需要实现类或者继承者去实现所有方法

不同点

1.抽象类不可以多重继承，接口可以(无论是多重类型继承还是多重行为继承);

2.抽象类和接口所反映出的设计理念不同。其实抽象类表示的是”is-a”关系，接口表示的是”like-a”关系;

3.接口中定义的变量默认是public static final 型，且必须给其初值，所以实现类中不能重新定义，也不能改变其值;抽象类中的变量默认是friendly 型，其值可以在子类中重新定义，也可以重新赋值。

三、多重继承的冲突说明

由于同一个方法可以从不同接口引入，自然而然的会有冲突的现象，默认方法判断冲突的规则如下：

1.一个声明在类里面的方法优先于任何默认方法(classes always win)

2.否则，则会优先选取最具体的实现，比如下面的例子 B重写了A的hello方法。

四、如何扩展或实现带有默认方法的接口?

当前扩展一个默认方法的接口时，你可以采用以下三种方式：

1：让扩展类继承默认方法，根据不管是否存在默认方法

2：重新声明默认方法，使其变为一个抽象方法（注意，扩展类的实现类必须实现此方法）

3：重新定义默认方法，覆盖(override)父类的默认方法

五、默认方法样例代码

```
import java.time.DateTimeException;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;

public interface TimeClient {
    void setTime(int hour, int minute, int
second);

    void setDate(int day, int month, int year);

    void setDateAndTime(int day, int month, int
year, int hour, int minute,
                        int second);

    LocalDateTime getLocalDateTime();

    static ZoneId getZoneId(String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid
time zone: " + zoneString
                               + "; using
default time zone instead.");
            return ZoneId.
systemDefault();
        }
    }

    default ZonedDateTime
getZonedDateTime(String zoneString) {
        return ZonedDateTime.
of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```

----------第二段----------

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.util.ArrayList;
```

```
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class SimpleTimeClient implements
TimeClient {

    private LocalDateTime dateAndTime;

    public SimpleTimeClient() {
        dateAndTime = LocalDateTime.now();
    }

    public void setTime(int hour, int minute, int
second) {
        LocalDate currentDate = LocalDate.
from(dateAndTime);
        LocalTime timeToSet = LocalTime.
of(hour, minute, second);
        dateAndTime = LocalDateTime.
of(currentDate, timeToSet);
    }

    public void setDate(int year, int month, int day)
{
        LocalDate dateToSet = LocalDate.
of(year, month, day);
        LocalTime currentTime = LocalTime.
from(dateAndTime);
        dateAndTime = LocalDateTime.
of(dateToSet, currentTime);
    }

    public void setDateAndTime(int year, int
month, int day, int hour,
            int minute, int second) {
        LocalDate dateToSet = LocalDate.
of(year, month, day);
        LocalTime timeToSet = LocalTime.
of(hour, minute, second);
        dateAndTime = LocalDateTime.
of(dateToSet, timeToSet);
    }

    public LocalDateTime getLocalDateTime() {
        return dateAndTime;
    }

    public String toString() {
        return dateAndTime.toString();
    }

    public static void main(String... args) {
        TimeClient client = new
SimpleTimeClient();
        // 显示当前日期时间
        System.out.println(client.toString());
        // 设置日期
        client.setTime(11, 12, 22);
        System.out.println(client);
        // 设置时间
```

```
        client.setDate(2012, 11, 12);
        System.out.println(client);

        System.out.println("Time in Asia/
Shanghai: "
                + client.
getZonedDateTime("Asia/Shanghai").toString());
    }
}
```

六、整合默认方法、静态方法到已经存在的接口

默认方法使您能够添加新的功能到已经存在的接口，确保与采用老版本这些接口编写的代码的二进制兼容性。特别是，默认的方法使您能够在已经存在的接口中添加使用lambda表达式作为参数的方法。下面的样例代码说明通过默认方法和静态方法，Comparator 接口是如何提供丰富的功能的。

在java8中，Compartor接口提供了丰富的功能，提供了差不多近20个默认或静态方法，在以前的版本中仅仅提供了compare(T o1, T o2)一个比较接口方法

下面的代码是有关扑克牌游戏中的洗牌，针对牌排序，打散，发牌的部分源代码

```
package defaultmethods;

//扑克牌接口类
public interface Card extends Comparable<Card> {

public enum Suit {
   DIAMONDS (1, "Diamonds"),
   CLUBS   (2, "Clubs"   ),
   HEARTS   (3, "Hearts"  ),
   SPADES   (4, "Spades"  );

   private final int value;
   private final String text;
   Suit(int value, String text) {
      this.value = value;
      this.text = text;
   }
   public int value() {return value;}
   public String text() {return text;}
}

public enum Rank {
   DEUCE  (2 , "Two"  ),
   THREE  (3 , "Three"),
   FOUR   (4 , "Four" ),
   FIVE   (5 , "Five" ),
   SIX    (6 , "Six"  ),
   SEVEN  (7 , "Seven"),
   EIGHT  (8 , "Eight"),
   NINE   (9 , "Nine" ),
   TEN    (10, "Ten"  ),
   JACK   (11, "Jack" ),
   QUEEN  (12, "Queen"),
   KING   (13, "King" ),
   ACE    (14, "Ace"  );
```

```
        private final int value;
        private final String text;
        Rank(int value, String text) {
            this.value = value;
            this.text = text;
        }
        public int value() {return value;}
        public String text() {return text;}
    }

    public Card.Suit getSuit();
    public Card.Rank getRank();
}
```

----------第二段----------

```
package defaultmethods;

import java.util.Comparator;
import java.util.List;
import java.util.Map;

//牌桌接口类
public interface Deck {

    List<Card> getCards();
    Deck deckFactory();
    int size();
    void addCard(Card card);
    void addCards(List<Card> cards);
    void addDeck(Deck deck);
    void shuffle();
    void sort();
    void sort(Comparator<Card> c);
    String deckToString();

    Map<Integer, Deck> deal(int players, int
numberOfCards)
        throws IllegalArgumentException;
}
```

----------第三段----------

```
package defaultmethods;

import java.util.Comparator;

//先根据rank,再根据suit进行比较
public class SortByRankThenSuit implements
Comparator<Card> {
        public int compare(Card firstCard, Card
secondCard) {
                int compVal = firstCard.getRank().
value()
                                - secondCard.
getRank().value();
                if (compVal != 0)
                        return compVal;
                else
                        return firstCard.getSuit().
value() - secondCard.getSuit().value();
        }
}
```

```
------------------------------------------------------------
------------第四段-----------------------------------------
package defaultmethods;

//扑克牌实现类
public class PlayingCard implements Card {

    private Card.Rank rank;
    private Card.Suit suit;

    public PlayingCard(Card.Rank rank, Card.Suit
suit) {
        this.rank = rank;
        this.suit = suit;
    }

    public Card.Suit getSuit() {
        return suit;
    }

    public Card.Rank getRank() {
        return rank;
    }

    public boolean equals(Object obj) {
    if (obj instanceof Card) {
    if (((Card) obj).getRank() == this.rank
    && ((Card) obj).getSuit() == this.suit) {
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    }

    public int hashCode() {
        return ((suit.value() - 1) * 13) + rank.
value();
    }

    //实现比较接口
    public int compareTo(Card o) {
        return this.hashCode() -
o.hashCode();
    }
    //重载toString
    public String toString() {
    return this.rank.text() + " of " + this.suit.text();
    }

public static void main(String... args) {
new PlayingCard(Rank.ACE, Suit.DIAMONDS);
new PlayingCard(Rank.KING, Suit.SPADES);
    }
}
------------------------------------------------------------
------------第五段-----------------------------------------
package defaultmethods;
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

//牌桌实现类
public class StandardDeck implements Deck {

    //扑克牌列表
    private List<Card> entireDeck;

    public StandardDeck(List<Card> existingList) {
        this.entireDeck = existingList;
    }

    public StandardDeck() {
    this.entireDeck = new ArrayList<>();
    for (Card.Suit s : Card.Suit.values()) {
    for (Card.Rank r : Card.Rank.values()) {
        this.entireDeck.add(new PlayingCard(r,
s));
                }
            }
    }

    public Deck deckFactory() {
        return new StandardDeck(new
ArrayList<Card>());
    }

    public int size() {
        return entireDeck.size();
    }
    public List<Card> getCards() {
        return entireDeck;
    }

    public void addCard(Card card) {
        entireDeck.add(card);
    }

    public void addCards(List<Card> cards) {
        entireDeck.addAll(cards);
    }

    public void addDeck(Deck deck) {
        List<Card> listToAdd = deck.
getCards();
        entireDeck.addAll(listToAdd);
    }

    public void sort() {
        Collections.sort(entireDeck);
    }

    public void sort(Comparator<Card> c) {
```

```
            Collections.sort(entireDeck, c);
        }

        public void shuffle() {
            Collections.shuffle(entireDeck);
        }

//为每位玩家分牌
public Map<Integer, Deck> deal(int players, int
numberOfCards)
throws IllegalArgumentException {
int cardsDealt = players * numberOfCards;
int sizeOfDeck = entireDeck.size();

if (cardsDealt > sizeOfDeck) {
throw new IllegalArgumentException("Number of
players (" + players
+ ") times number of cards to be dealt (" +
numberOfCards
+ ") is greater than the number of cards in the deck
("
            + sizeOfDeck + ").");
            }
            //把牌分成几份
            int slices=players+1;
            if(cardsDealt == sizeOfDeck)
                slices=players;

            //根据玩家的个数，每个玩家分到的扑
克牌数进行分牌
            Map<Integer, List<Card>> dealtDeck
= entireDeck.stream().collect(
            Collectors.groupingBy(card -> {
        int cardIndex = entireDeck.indexOf(card);
            if (cardIndex >= cardsDealt)
                return (players + 1);
            else
            return (cardIndex % players) + 1;
                }));
        System.out.println(dealtDeck);
        // Convert Map<Integer, List<Card>> to
Map<Integer, Deck>
        Map<Integer, Deck> mapToReturn = new
HashMap<>();

        for (int i = 1; i < (slices + 1); i++) {
        Deck currentDeck = deckFactory();
        currentDeck.addCards(dealtDeck.get(i));
        mapToReturn.put(i, currentDeck);
            }
            return mapToReturn;
        }
public String deckToString() {
            return this.entireDeck.stream().
map(Card::toString)
            .collect(Collectors.joining("\n"));
        }

        public String toString(){
            return deckToString();
```

```
    }

    public static void main(String... args) {
        System.out.println("Creating deck:");
        StandardDeck myDeck = new StandardDeck();
        myDeck.sort();
        System.out.println("Sorted deck");
        System.out.println(myDeck.deckToString());

    myDeck.shuffle();
    myDeck.sort(new SortByRankThenSuit());
    System.out.println("Sorted by rank, then by suit");
    System.out.println(myDeck.deckToString());
    myDeck.shuffle();
    myDeck.sort(Comparator.comparing(Card::getRank).thenComparing(
    Comparator.comparing(Card::getSuit)));
    System.out.println("Sorted by rank, then by suit "
    + "with static and default methods");
System.out.println(myDeck.deckToString());

myDeck.sort(Comparator.comparing(Card::getRank).reversed()

.thenComparing(Comparator.comparing(Card::getSuit).reversed()));
System.out.println("Sorted by rank reversed, then by suit "
+ "with static and default methods");
System.out.println(myDeck.deckToString());

myDeck.shuffle();
myDeck.sort(
 (firstCard, secondCard) ->
 firstCard.getRank().value() - secondCard.getRank().value()
        );
        System.out.println(myDeck.deckToString());

        myDeck.shuffle();
        myDeck.sort(Comparator.comparing(Card::getRank));
        System.out.println(myDeck.deckToString());

Map<Integer, Deck> map=myDeck.deal(4, 11);
for(Map.Entry<Integer, Deck> item:map.entrySet()){
    System.out.println(item.getKey());
    System.out.println(item.getValue());
    System.out.println("------------------------------");
        }
    }
}
```

■

Java8使用Map中的computeIfAbsent方法构建本地缓存

java8在接口Map中增加了computeIfAbsent方法，可以通过此方法构建本地缓存，降低程序的计算量，程序的复杂度，使代码简洁，易懂。

一、概念及使用介绍

在JAVA8的Map接口中，增加了一个方法computeIfAbsent，此方法签名如下：

```
public V computeIfAbsent(K key, Function<?
super K,? extends V> mappingFunction)
```

Map接口的实现类如HashMap,ConcurrentHashMap,HashTable等继承了此方法，通过此方法可以构建JAVA本地缓存，降低程序的计算量，程序的复杂度，使代码简洁，易懂。

此方法首先判断缓存MAP中是否存在指定key的值，如果不存在，会自动调用mappingFunction(key)计算key的value，然后将key = value放入到缓存Map,java8会使用thread-safe的方式从cache中存取记录。

如果mappingFunction(key)返回的值为null或抛出异常，则不会有记录存入map

二 代码样例

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.concurrent.
ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Main {
    static Map<Integer, Integer> cache = new
ConcurrentHashMap<>();

    public static void main(String[] args)
throws InterruptedException {
        cache.put(0, 0);
        cache.put(1, 1);
        // 普通方式
        System.out.println("Fibonacci(7) = "
+ fibonacci(7));
        // 采用java7的同步线程方式及java8
的本地缓存的方式
        System.out.
println("FibonacciJava8(7) = " +
fibonacciJava8(7));
        System.out.
println("FibonacciJava7(7) = " +
```

```
fibonacciJava7(7));

        // 构建多值Map样例代码
        Map<String, HashSet<String>>
map1 = new HashMap<>();
        map1.computeIfAbsent("fruits", k ->
genValue(k)).add("apple");
        map1.computeIfAbsent("fruits", k ->
genValue(k)).add("orange");
        map1.computeIfAbsent("fruits", k ->
genValue(k)).add("pear");
        map1.computeIfAbsent("fruits", k ->
genValue(k)).add("banana");
        map1.computeIfAbsent("fruits", k ->
genValue(k)).add("water");
        System.out.println(map1);

        //测试多线程并发处理，是否同步操
作
        Map<String, String> map2 = new
ConcurrentHashMap<>();
        ExecutorService exec = Executors.
newCachedThreadPool();
        for (int i = 0; i < 5; i++) {
            exec.execute(() -> {
                map2.
computeIfAbsent("name", k -> genValue2(k));
                map2.
computeIfAbsent("addr", k -> genValue2(k));
                map2.
computeIfAbsent("email", k -> genValue2(k));
                map2.
computeIfAbsent("mobile", k -> genValue2(k));
            });
        }
        exec.shutdown();
        exec.awaitTermination(1,
TimeUnit.SECONDS);
        System.out.println(map2);
    }

    static HashSet<String> genValue(String
str) {
        return new HashSet<String>();
    }

    static String genValue2(String str) {
        System.out.println("===");
        return str + "2";
    }

    /**
     * 普通的实现方式 普通方式使用大量的
计算，存在性能问题. 并且计算量随着n的增加
呈指数级增加，需要用到一些缓存策略，并且
是线程安全的.
     *
     * @param n
     * @return
```

```
     */
    static int fibonacci(int n) {
        if (n == 0 || n == 1)
            return n;
        System.out.println("calculating Fibonacci(" + n + ")");
        return fibonacci(n - 2) + fibonacci(n - 1);
    }

    /**
     * 采用java8的本地缓存方式 如果缓存MAP中不存在指定key的值，会自动调用mappingFunction(key)计算key的value
     * 然后将key = value放入到缓存Map,java8会使用thread-safe的方式从cache中存取记录
     *
     * @param n
     * @return
     */
    static int fibonacciJava8(int n) {
        return cache.computeIfAbsent(n, (key) -> {
            System.out.println("calculating FibonacciJava8 " + n);
            return fibonacciJava8(n - 2) + fibonacciJava8(n - 1);
        });
    }

    /**
     * 在java7中的实现方式
     * 在java7中，通过synchronized进行线程同步，检查缓存是否存在key对应的值，如果不存在才进行计算并放入缓存中
     * 为了更好的性能，需要使用 double-checked locking，那样代码会更复杂
     *
     * @param n
     * @return
     */
    static int fibonacciJava7(int n) {
        if (n == 0 || n == 1)
            return n;

        Integer result = cache.get(n);

        if (result == null) {
            synchronized (cache) {
                result = cache.get(n);

                if (result == null) {
                    System.out.println("calculating FibonacciJava7(" + n + ")");
                    result = fibonacciJava7(n - 2) + fibonacciJava7(n - 1);
                    cache.put(n, result);
                }
            }
```

```
            }
            return result;
        }
    }
```

三，程序运行结果

```
calculating Fibonacci(7)
calculating Fibonacci(5)
calculating Fibonacci(3)
calculating Fibonacci(2)
calculating Fibonacci(4)
calculating Fibonacci(2)
calculating Fibonacci(3)
calculating Fibonacci(2)
calculating Fibonacci(6)
calculating Fibonacci(4)
calculating Fibonacci(2)
calculating Fibonacci(3)
calculating Fibonacci(2)
calculating Fibonacci(5)
calculating Fibonacci(3)
calculating Fibonacci(2)
calculating Fibonacci(4)
calculating Fibonacci(2)
calculating Fibonacci(3)
calculating Fibonacci(2)
Fibonacci(7) = 13
calculating FibonacciJava8 7
calculating FibonacciJava8 5
calculating FibonacciJava8 3
calculating FibonacciJava8 2
calculating FibonacciJava8 4
calculating FibonacciJava8 6
FibonacciJava8(7) = 13
FibonacciJava7(7) = 13
{fruits=[orange, banana, apple, pear, water]}
===
===
===
===
{name=name2, mobile=mobile2, addr=addr2,
email=email2}
```

四、参考

http://www.java8.org/caching-with-ConcurrentHashMap-in-java-8.html

JDK8 API

http://stackoverflow.com/questions/19278443/how-do-i-use-the-new-computeifabsent-function

■

Java 8 的默认方法和多重继承

在我通读文档，并写下一些示例程序来确保自己对这些新的特性有一个良好的理解是，有一个特性引起了我特别的注意: 默认方法，首先，让我们来看一看它们到底是什么。

介绍

我一看到 Java 8 发布了，就决定开始深入看看对于这门编程语言新的增强功能。 我承认，自己并没有像对待 7 那样对这个版本进行跟进，所以我所知道的唯一的增强功能只是lambda表达式而已。在我通读文档，并写下一些示例程序来确保自己对这些新的特性有一个良好的理解是，有一个特性引起了我特别的注意: 默认方法，首先，让我们来看一看它们到底是什么。

先认识认识

从本质上讲，默认方法就是一个在接口里面有了一个实现的方法。 传统的接口只是简单的由抽象方法和公共静态的并且是final的变量构成。 那么考虑看看下面这个:

```
public interface ExampleInterface {
    public void exampleMethod();
    default public void exampleDefaultMethod() {
        // Some implementation code here
    }
}
```

如果一个类决定去实现 ExampleInterface，它只需要去实现 exampleMethod，并且它可以简单地采用 exampleDefaultMethod的默认实现。 现在，让我们来看看Oracle打算让我们怎样去使用它们。

它们应该怎么被使用

这个想法不是让你在设计时考虑使用默认方法作为解决方案，而是它们只是一个事后的考虑。 在Oracle的跟踪报道上他们称述的意图是去“确保为那些老版本接口写的代码的二进制兼容性”(1)。从本质上讲，你和你的开发人员编写了对某些接口的多个实现，但是你想要添加新的一块东西。 然而，这样做就意味着对所有那些实现了这个接口的类进行重新构建都将失败，除非你为它们每一个都补上新增的那一块的实现。 默认方法的想法是你可以在不改变现有实现并且再也不用变更它们的前提下加入对接口的新需求。

但是让我们退后一步想想。 这里面还是有些问题的吧。 首先，让我们接受有时高级别 的变更(比如变更你的接口)行将发生的这一事实。 然而，让我们也不要忘了接口的存在的目的是什么。 理想情况下一个接口锁定一些高级别的对象，或者说你是在创造一些特殊功能部分的API。 如果一些类只需要去实现接口功能 的某些部分，而不是全部的话，那么那会是一个好的设计吗？那不就是抽象类被设计出来的原因么？

其次，这会引入对于处理多重继承的需求。 而现在谢天谢地，如果你有两个拥有一个对于单个方法签名的默认实现的接口，而你尝试去拥有一个同时实现这两个接口的类， Java将拒绝编译这个类。许多人抱怨接口中允许公共静态变量的不确定性，说如果两个接口中有同一个变量就会有些模棱两可(因此像 C# 这样的编程语言就会禁止诸如此类的事情)，而这是有一些难以对付的。

我看到这个功能是如何在这个问题上起作用的，而我不准备在这里阐述它，直到我看到它被应用在实践中，但是我必须强调它使我感到紧张。 话虽这么说，让我们还是来更多的看看这个功能可能会被运用到的地方，而不是它的这个目的。

可能的误用

前面我们谈到引入多重继承后模棱两可的问题，那么编译器不会捕捉到的会是什么情况呢？什么才会使我停止像我使用一个抽象类(通常是一些抽象方法和一些实现)那样,简单的使用接口的做法，而去利用继承多个接口的好处呢？这没有怎么不好，但是究其自身而言，它将会鼓励使用Java努力避免的复杂设计。 有些情况是在你实际需要多继承来解决你在构建一个抽象类的框架中使用单个继承类型所不能解决的问题。

总结

总的来说，Java8还是提供了许多很酷的新东西的，有些看起来同Java早期的基础知识相悖。不管是好是坏，时间会证明一切。 如果你有任何建议，请自由的分享他们。 此外，任何时候开始在你的项目中使用Java 8，请自由分享你的成功或恐怖故事。 可以通过ryan。■

编者按

Java 8终于到来了！经过几年的等待，java程序员终于能在java中得到函数式编程的支持了．函数式编程的支持能流程化现有的代码并且为java提供强大的能力……

Java 8 彻底改变数据库访问

Java 8终于到来了！经过几年的等待，java程序员终于能在java中得到函数式编程的支持了．函数式编程的支持能流程化现有的代码并且为java提供强大的能力.在这些新特性中最瞩目的是java程序员对数据库的操作方式.函数式编程带来了令人激动的简便高效的数据库API．Java 8 将会支持可与C#，LINQ等语言竞争的新的数据库访问方式.

处理数据的函数式方式

Java 8 不仅仅添加了函数式支持,它也通过新的函数式处理数据的方式扩展了集合(Collection)类．而通常情况下java处理大量数据时需要大量的循环和迭代器.

例如，假设你有一个存储客户(Customer)对象的collection:

```
Collection<Customer>&amp;nbsp;customers;
```

如果你只对来自Belgium的客户感兴趣，你将不得不迭代所有的customer对象并只保存你需要的.

```
Collection<Customer> belgians = new
ArrayList<>();
for (Customer c : customers) {
  if (c.getCountry().equals(“Belgium”))
    belgians.add(c);
}
```

这不仅花费了5行代码,而且它也不怎么抽象.假使你有1千万个对象时会怎样呢?你会通过两个线程并发过滤所有对象来提速么?那你将不得不使用大量危险的多线程代码来重写所有代码.

而通过Java 8,仅仅只需要一行代码就能实现相同的功能.通过对函数式编程的支持，Java 8 能让你只写一个函数表明你对哪些客户(对象)感兴趣然后使用那个函数对集合做过滤就可以了．Java 8 的新Steams API 支持你这样做:

```
customers.stream().filter(
  c -> c.getCountry().equals(“Belgium”)
);
```

上面Java 8 版本的代码不仅更短,而且更容易理

解.它几乎没有什么 陈词滥调(循环或迭代器等).代码调用了filter()方法,那很明显这段代码是用来过滤客户(对象)的.你不需要再把时间浪费在解读循环中的代码来理解它在对它的数据做什么.

假使你想并发执行这段代码该怎么办呢?你只需使用另一个类型的stream

```
customers.parallelStream().filter(
  c -> c.getCountry().equals(“Belgium”)
);
```

更另人激动的是这种函数式风格的代码也同样适用于数据库

在数据库上使用函数式方式

传统上来说，程序员需要用特殊数据库查询语句去访问数据库的数据. 例如,下面就是用 JDBC 代码去查找来自Belgium的客户:

```
PreparedStatement s = con.prepareStatement(
    "SELECT * "
  + "FROM Customer C "
  + "WHERE C.Country = ? ");
s.setString(1, "Belgium");
ResultSet rs = s.executeQuery();
```

大部分这些代码都是字符串，这样会使编译器不能发现错误而且这草率的代码会导致安全问题. 还有这些大量的样板代码使得写数据访问代码变得十分冗余. 一些工具例如 jOOQ ， 通过使用特殊的java库去提供数据库查询语言可以解决错误检查和安全问题。 或者使用对象关系映射工具可以免去大量的无趣的代码，可它们只能用在通用访问查询，如果需要复杂的查询，还是需要用特殊的数据库查询语言。

使用Java 8,借助流式API就可以用函数式方式去查询数据库了。例如，Jinq 是一个开源的项目，它探索怎样的未来数据库API可以令函数式编程成为可能。这里就是一个使用Jinq的数据库查询:

```
customers.where(
  c -> c.getCountry().equals("Belgium")
);
```

这代码几乎跟跟使用流式API的代码一样. 事实上，未来的Jinq版本可以让你用流式API直接写数据库查询。 当代码运行的时候，Jinq将自动翻译成数据库查询代码，正如之前JDBC查询一样。

这样的话，就算没有学过一些新的数据库查询语言，你也可以写出有效率的数据库查询。你可以用同样样式的代码用在java集合上。你也不需要特殊的java编译器或者虚拟机。所有的代码编译和运行在普通的java 8 JDK上。如果你的代

码有错误，编译器将找出它们并且报告给你，就像普通的java代码。

Jinq 支持跟SQL92一样的复杂查询. Selection（选择），projection（投影），joins（连接），和子查询 它都支持。翻译java代码成数据库查询的算法是十分灵活的，只要是它能接受的，都能翻译。例如，Jinq能够翻译下面的数据库查询，尽管它很复杂。

```
customers
  .where( c -> c.getCountry().equals(“Belgium”) )
  .where( c -> {
    if (c.getSalary() < 100000)
      return c.getSalary() < c.getDebt();
    else
      return c.getSalary() < 2 * c.getDebt();
    } );
```

正如你看到的，java 8 的函数式编程非常适合数据库查询。而且查询紧凑，甚至复杂的查询也能够胜任。

内部运作

但这都是如何工作的呢？怎么能让普通的Java编译器将Java代码转换成数据库查询？Java 8 有什么特别之处使这个成为可能？

支持这些函数性风格的新的数据库PI的关键是一种叫做“象征性执行”的字节码分析手段。虽然你的代码是被一个普通的Java编译器编译的并运行在一个普通的Java虚拟机中，但 Jinq 能够在你被编译的Java代码运行时进行分析并从中构建数据库查询。使用 Java 8 Streams API 时，常会发现分析短小的函数时，象征性执行的工作效果最好。

要了解这个象征性执行是如何工作的，最简单的方法是用一个例子。让我们检查一下下面的查询是如何被 Jinq 转换为SQL查询语言的：

```
customers
  .where( c -> c.getCountry().
equals(“Belgium”) )
```

初始时，变量 customers 是一个集合，其对应的数据库查询是：

```
SELECT *
 FROM Customers C
```

然后，where() 方法被调用，一个函数被传递给它。在 where() 方法中，Jinq 打开这个函数的 .class 文件，得到这个函数被编译成的字节码进行分析。在这个例子中，不使用真正的字节码，让我们用一些简单的指令来代表这个函数的字节码：

```
1.d = c.getCountry()
2.e = &#8220;Belgium&#8221;
3.e = d.equals(e)
```

4.return e

在这里，我们假设函数已被Java编译器编译成这四条指令。当调用 where() 方法时，Jinq 看到的就是这些。如何才能使Jinq理解这些代码呢?

Jinq 通过执行代码来分析。但 Jinq 不直接运行代码。它是“抽象”地运行代码：不使用真实的变量和真实的值，Jinq 使用符号来表示执行代码时的所有值。这就是这个分析为什么被称为“象征性执行”。

Jinq 执行每条指令，并跟踪所有的副作用或代码在程序状态时改变的所有东西。下面是一个图表，显示出 Jinq 用象征性执行方式执行这四行代码时发现的所有副作用。

象征性执行的例子

在图中，你可以看到第一条指令运行后，Jinq 发现了两个副作用：变量d已经发生了变化，方法 Customer.getCountry() 被调用。由于是象征性执行，变量d没有给出一个真正的比如是“USA”或“Denmark”的值，它被分配为 c.getCountry() 的象征性的值。

在所有这些指令被象征性执行之后，Jinq 对副作用作精简。由于变量 d 和 e 是局部变量，它们的任何变化在函数退出后都会被丢弃，所以这些副作用可以忽略不计。Jinq也知道 Customer.getCountry() and String.equals() 方法没修改任何变量或显示任何输出，因此这些方法调用也可以被忽略。由此，Jinq 可以得出这样的结论：执行这个函数只会产生一个作用，它会返回 c.getCountry().equals(“Belgium”)。

一旦Jinq已明白在 where()方法中传递给它的函数，它可以混合数据库查询方面的知识，优先于 customers 集合来创建一个新的数据库查询。

生成数据库查询

这就是 Jinq 如何从你的代码生成数据库查询的。象征性执行的使用意味着，这种方法对于不同的Java编译器输出的不同的代码模式都是相当强大的。如果 Jinq 遇到的代码有不能转化为数据库查询的副作用，Jinq 将保持你的这些代码不变。因为一切都是用正常的Java代码写的，Jinq 可以直接运行那些代码，您的代码将产生预期的结果。

这个简单的翻译实例应该让你明白了怎样查询翻译作品。你可以确信，这些算法可以正确地从你的代码生成数据库查询。

美好前景

我希望我已经让你品尝到了Java 8带来的在Java中进行数据库工作的新方式。Java 8 支持的函数式编程允许你用和为Java集合编写代码同样的方式来为数据库写代码。希望不久现有的数据库API都能被扩展以支持这些类型的查询。■

Java 8 Nashorn 脚本引擎

□ 架构梦想 /文

> "Java8 终于来了.函数式接口,lambda表达式期待很久了.新的武器在手,应该可以玩出新花样.
>
> 前两天无意中发现Java8 中还带了另外一个有意思的东西.
>
> Nashorn 一个Javascript引擎.
>
> 这等好玩的东西不把玩一下实在是浪费了.所以直接找到了oracle官方的介绍文档.说实话,文档真给力啊."

这篇文章是我一边看Oracle官方文档,一边敲代码试验,一边写的,不算是翻译的文档,算是中文版总结文档吧.呵呵.

原文地址:Oracle Nashorn: A Next-Generation JavaScript Engine for the JVM

java7以前,JDK内置了一个基于Mozilla Rhino的javascript脚本引擎.在java8里面,基于JSR292和invokedynamic重新提供了一个新的javascript引擎-Oracle Nashorn.它更符合ECMA标准的javascript规范,而且基于invokedynamic调用拥有更好的性能.

文章使用的是最新JDK8.所以想用要先装一下.

·第一个栗子

第一个程序一定是HelloWorld.而且是命令行下面的实现.因此.打开一个命令行吧.

如果你的命令行配好了,输入 jjs 回车

就可以看到Nashrn的命令行了.

print(“hello halu”);

如果你人品没问题,你一定看到输出了…../抠鼻

·第二个栗子

命令行下面可以用了,那么抓紧试试用js文件吧.

把下面的代码保存在一个文件里面,我的文件名是halu.js

```
function SayHi(){
  print(“hello halu”);
}
SayHi();
```

然后打开命令行,cd到文件所在的目录.

windows下面cd有个技巧可以用.在文件夹空

白的位置按住shift右击鼠标,菜单中会出现在此处打开命令窗口的选项,一般人我可不告诉他.

命令行下输入 jjs halu.js 就可以看到执行结果了.

・第三个栗子

要知道,这可是java8 环境.看下面的例子吧.

```
var data = [1,2,3,4,5,6,7,8,9,10];
var filtered = data.filter(function(i){
    return i%2 == 0;
},0);
print(filtered);
var sumOfFilterd = filtered.reduce(function
(acc,next){
return acc + next;
  },0);
print(sumOfFilterd);
```

看一下执行结果吧.这个例子里面信息量可是巨大的.

Nashorn 只是使用遵从 ECMA 规范的javascript语言,在网页上常用的对象Nashorn里面并没有.比如说 console,window等对象.

・第四个栗子

命令行执行以下 jjs -help,在帮助中可以看到.

・jjs可以运行javaFX程序脚本

・可以使用javascript严格模式.

・可以指定额外的classpath.

・一个有趣的脚本模式

脚本模式很有趣,你可以使用jjs运行使用javascript编写的系统脚本.类似python,ruby,bash脚本.脚本模式有两种扩展:heredocs和shell invocations.

```
var data = {
    foo:"bar",
    time: new Date()
};
print("< So...foo = ${data.foo} and the current
time is ${data.time} EOF");
```

使用 jjs -scripting halu.js 执行该脚本.

heredocs 是一种简单的多行文本,使用类似bash的语法.使用 < 符号开始后面跟一个特殊标记.字符串中可以使用 ${} 表达式(类似EL表达式).需要注意的是,如果使用单引号引起来的字符串,内部的标示是不会被替换的.

・第五个栗子

Shell invocations 是允许调用命令行程序.

```
var lines =\`ls`;
print(lines);
```

这样就可以执行shell命令了.当然windows下面失败….

*注意符号是 ` [波浪线那个键],不是 ‘ [单引号] * ,我找了好久才发现.

・第六个栗子

下面我们来写一个 java 程序

```
import javax.script.ScriptEngine;
 import javax.script.ScriptEngineManager;
 public class halu{
    public static void main(String[] args) throws Throwable{
        ScriptEngineManager engineManager = new ScriptEngineManager();
        ScriptEngine engine = engineManager.getEngineByName("nashorn");
        engine.eval("function sum(a,b){return a+b;}");
        System.out.println(engine.eval("sum(1,2);"));
    }
 }
```

这段代码很简单,使用ScriptEngineManager获得一个 ScriptEngine.然后通过eval函数执行字符串脚本.

```
Invocable invocable = (Invocable) engine;
 System.out.println(invocable.invokeFunction("sum",1,3));
```

engine 也可以使用invocable接口调用脚本内的函数.

增加一个 Adder.java 的接口

```
public interface Adder{
    int sum(int a, int b);
}
```

halu类里面

```
Adder adderaa = invocable.getInterface(Adder.class);
 System.out.println(""+ adderaa.sum(2,3));
```

这样可以将javascript的类映射到java的接口上.个人感觉这是个很强大的功能.

读取脚本文件执行.使用 java.io.FileReader;

```
engine.eval(new FileReader("halu.js"));
```

• 第七个栗子

我们来使用javascript调用java

```
print(java.lang.System.currentTimeMillis());
 var file = new java.io.File("halu.js");
 print(file.getAbsolutePath());
 print(file.absolutePath);
```

使用 jjs 命令执行哦~

集合的使用

```
var stack =
 new java.util.LinkedList();
 [1, 2, 3, 4].forEach(function(item) {
    stack.push(item);
 });
 print(stack);
 print(stack.getClass());
```

·第八个栗子

javascript实现java接口

```
var iterator = new java.util.Iterator({
    i: 0,
    hasNext: function() {
        return this.i < 10;
    },
    next: function() {
        return this.i++;
    }
});
print(iterator instanceof Java.type("java.util.Iterator"));
while (iterator.hasNext()) {
    print("-> " + iterator.next());
}
```

·第九个栗子

javascript 实现多个接口

```
var ObjectType = Java.type("java.lang.Object");
var Comparable = Java.type("java.lang.Comparable");
var Serializable = Java.type("java.io.Serializable");
var MyExtender = Java.extend(
    ObjectType, Comparable, Serializable);
var instance = new MyExtender({
    someInt: 0,
    compareTo: function(other) {
        var value = other["someInt"];
        if (value === undefined) {
            return 1;
        }
        if (this.someInt < value) {
            return -1;
        } else if (this.someInt == value) {
            return 0;
        } else {
            return 1;
        }
    }
});
print(instance instanceof Comparable);
print(instance instanceof Serializable);
print(instance.compareTo({ someInt: 10 }));
print(instance.compareTo({ someInt: 0 }));
print(instance.compareTo({ someInt: -10 }));
```

终于完了….累死我了….

感受

Nashorn 真心好玩,javascript语言本身就有无限可能性,应该能做出来一些好玩的东西.下面就该考虑如何使用Nashorn做点东西了.

PS: shell invocable 不支持 windows 真心觉得坑…..谁要是有办法请留言给我.■

Java并没没落：最新Java 8简明教程

■ 编者按

欢迎阅读我编写的Java 8介绍。本教程将带领你一步一步地认识这门语言的新特性。通过简单明了的代码示例，你将会学习到如何使用默认接口方法，Lambda表达式，方法引用和重复注解。看完这篇教程后，你还将对最新推出的API有一定的了解。

ImportNew注：有兴趣第一时间学习Java 8的Java开发者，欢迎围观《征集参与Java 8原创系列文章作者》。

以下是《Java 8简明教程》的正文。

“Java并没有没落，人们很快就会发现这一点”

欢迎阅读我编写的Java 8介绍。本教程将带领你一步一步地认识这门语言的新特性。通过简单明了的代码示例，你将会学习到如何使用默认接口方法，Lambda表达式，方法引用和重复注解。看完这篇教程后，你还将对最新推出的API有一定的了解，例如：流控制，函数式接口，map扩展和新的时间日期API等等。

允许在接口中有默认方法实现

Java 8 允许我们使用default关键字，为接口声明添加非抽象的方法实现。这个特性又被称为扩展方法。下面是我们的第一个例子：

```
interface Formula {
    double calculate(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

在接口Formula中，除了抽象方法caculate以外，还定义了一个默认方法sqrt。Formula的实现类只需要实现抽象方法caculate就可以了。默认方法sqrt可以直接使用。

```
Formula formula = new Formula() {
    @Override
    public double calculate(int a) {
        return sqrt(a * 100);
    }
};

formula.calculate(100);     // 100.0
formula.sqrt(16);           // 4.0
```

formula对象以匿名对象的形式实现了Formula接口。代码很啰嗦：用了6行代码才实现了一个简单的计算功能：a*100开平方根。我们在下一节会看到，Java 8 还有一种更加优美的方法，能够实现包含单个函数的对象。

Lambda表达式

让我们从最简单的例子开始，来学习如何对一个string列表进行排序。我们首先使用Java 8之前的方法来实现：

```
List<String> names = Arrays.asList("peter",
"anna", "mike", "xenia");

Collections.sort(names, new
Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
});
```

静态工具方法Collections.sort接受一个list，和一个Comparator接口作为输入参数，Comparator的实现类可以对输入的list中的元素进行比较。通常情况下，你可以直接用创建匿名Comparator对象，并把它作为参数传递给sort方法。

除了创建匿名对象以外，Java 8 还提供了一种更简洁的方式，Lambda表达式。

```
Collections.sort(names, (String a, String b) -> {
    return b.compareTo(a);
});
```

你可以看到，这段代码就比之前的更加简短和易读。但是，它还可以更加简短：

```
Collections.sort(names, (String a, String b) ->
b.compareTo(a));
```

只要一行代码，包含了方法体。你甚至可以连大括号对{}和return关键字都省略不要。不过这还不是最短的写法：

```
Collections.sort(names, (a, b) -> b.
compareTo(a));
```

Java编译器能够自动识别参数的类型，所以你就可以省略掉类型不写。让我们再深入地研究一下lambda表达式的威力吧。

函数式接口

Lambda表达式如何匹配Java的类型系统？每一个lambda都能够通过一个特定的接口，与一个给定的类型进行匹配。一个所谓的函数式接口必须要有且仅有一个抽象方法声明。每个与之对应的lambda表达式必须要与抽象方法的声明相匹配。由于默认方法不是抽象的，因此你可以在你的函数式接口里任意添加默认方法。

任意只包含一个抽象方法的接口，我们都可以用来做成lambda表达式。为了让你定义的接口满足要求，你应当在接口前加上@FunctionalInterface标注。编译器会注意到这个标注，如果你的接口中定义了第二个抽象方法的话，编译器会抛出异常。

举例：

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}

Converter<String, Integer> converter = (from)
-> Integer.valueOf(from);
```

```
Integer converted = converter.convert("123");
System.out.println(converted);    // 123
```

注意，如果你不写@FunctionalInterface 标注，程序也是正确的。

方法和构造函数引用

上面的代码实例可以通过静态方法引用，使之更加简洁：

```
Converter<String, Integer> converter =
Integer::valueOf;
Integer converted = converter.convert("123");
System.out.println(converted);   // 123
```

Java 8 允许你通过::关键字获取方法或者构造函数的的引用。上面的例子就演示了如何引用一个静态方法。而且，我们还可以对一个对象的方法进行引用：

```
class Something {
    String startsWith(String s) {
        return String.valueOf(s.charAt(0));
    }
}

Something something = new Something();
Converter<String, String> converter =
something::startsWith;
String converted = converter.convert("Java");
System.out.println(converted);    // "J"
```

让我们看看如何使用::关键字引用构造函数。首先我们定义一个示例bean，包含不同的构造方法：

```
class Person {
    String firstName;
    String lastName;

    Person() {}

    Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

接下来，我们定义一个person工厂接口，用来创建新的person对象：

```
interface PersonFactory<P extends Person> {
    P create(String firstName, String lastName);
}
```

然后我们通过构造函数引用来把所有东西拼到一起，而不是像以前一样，通过手动实现一个工厂来这么做。

```
PersonFactory<Person> personFactory =
Person::new;
Person person = personFactory.create("Peter",
"Parker");
```

我们通过Person::new来创建一个Person类构造函数的引用。Java编译器会自动地选择合适的构造函数来匹配PersonFactory.create函数的签名，并选择正确的构造函数形式。

Lambda的范围

对于lambdab表达式外部的变量，其访问权限的粒度与匿名对象的方式非常类似。你能够访问局部对应的外部区域的局部final变量，以及成员变量和静态变量。

访问局部变量

我们可以访问lambda表达式外部的final局部变量：

```
final int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);

stringConverter.convert(2);     // 3
```

但是与匿名对象不同的是，变量num并不需要一定是final。下面的代码依然是合法的：

```
int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);

stringConverter.convert(2);     // 3
```

然而，num在编译的时候被隐式地当做final变量来处理。下面的代码就不合法：

```
int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);
num = 3;
```

在lambda表达式内部企图改变num的值也是不允许的。

访问成员变量和静态变量

与局部变量不同，我们在lambda表达式的内部能获取到对成员变量或静态变量的读写权。这种访问行为在匿名对象里是非常典型的。

```
class Lambda4 {
    static int outerStaticNum;
    int outerNum;

    void testScopes() {
        Converter<Integer, String>
stringConverter1 = (from) -> {
            outerNum = 23;
            return String.valueOf(from);
        };

        Converter<Integer, String>
stringConverter2 = (from) -> {
            outerStaticNum = 72;
            return String.valueOf(from);
        };
    }
}
```

访问默认接口方法

还记得第一节里面formula的那个例子么？ 接口Formula定义了一个默认的方法sqrt，该方法能够访问formula所有的对象实例，包括匿名对象。这个对lambda表达式来讲则无效。

默认方法无法在lambda表达式内部被访问。因此下面的代码是无法通过编译的：

```
Formula formula = (a) -> sqrt( a * 100);
```

内置函数式接口

JDK 1.8 API中包含了很多内置的函数式接口。有些是在以前版本的Java中大家耳熟能详的，例如Comparator接口，或者Runnable接口。对这些现成的接口进行实现，可以通过@FunctionalInterface 标注来启用Lambda功能支持。

此外，Java 8 API 还提供了很多新的函数式接口，来降低程序员的工作负担。有些新的接口已经在Google Guava库中很有名了。如果你对这些库很熟的话，你甚至闭上眼睛都能够想到，这些接口在类库的实现过程中起了多么大的作用。

Predicates

Predicate是一个布尔类型的函数，该函数只有一个输入参数。Predicate接口包含了多种默认方法，用于处理复杂的逻辑动词（and，or，negate）

```
Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test(“foo”);              // true
predicate.negate().test(“foo”);     // false

Predicate<Boolean> nonNull = Objects::nonNull;
Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNotEmpty = isEmpty.negate();
```

Functions

Function接口接收一个参数，并返回单一的结果。默认方法可以将多个函数串在一起（compse, andThen）

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply(“123”);     // “123”
```

Suppliers

Supplier接口产生一个给定类型的结果。与Function不同的是，Supplier没有输入参数。

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get();   // new Person
```

Consumers

Consumer代表了在一个输入参数上需要进行的操作。

```
Consumer<Person> greeter = (p) -> System.out.
println("Hello, " + p.firstName);
greeter.accept(new Person("Luke",
"Skywalker"));
```

Comparators

Comparator接口在早期的Java版本中非常著名。Java 8 为这个接口添加了不同的默认方法。

```
Comparator<Person> comparator = (p1, p2) ->
p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");
Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2);             // > 0
comparator.reversed().compare(p1, p2);  // < 0
```

Optionals

Optional不是一个函数式接口，而是一个精巧的工具接口，用来防止NullPointerEception产生。这个概念在下一节会显得很重要，所以我们在这里快速地浏览一下Optional的工作原理。

Optional是一个简单的值容器，这个值可以是null，也可以是non-null。考虑到一个方法可能会返回一个non-null的值，也可能返回一个空值。为了不直接返回null，我们在Java 8中就返回一个Optional.

```
Optional<String> optional = Optional.
of("bam");

optional.isPresent();           // true
optional.get();                 // "bam"
optional.orElse("fallback");    // "bam"

optional.ifPresent((s) -> System.out.println(s.
charAt(0)));     // "b"
```

Streams

java.util.Stream表示了某一种元素的序列，在这些元素上可以进行各种操作。Stream操作可以是中间操作，也可以是完结操作。完结操作会返回一个某种类型的值，而中间操作会返回流对象本身，并且你可以通过多次调用同一个流操作方法来将操作结果串起来（就像StringBuffer的append方法一样————译者注）。Stream是在一个源的基础上创建出来的，例如java.util.Collection中的list或者set（map不能作为Stream的源）。Stream操作往往可以通过顺序或者并行两种方式来执行。

我们先了解一下序列流。首先，我们通过string类型的list的形式创建示例数据：

```
List<String> stringCollection = new
ArrayList<>();
stringCollection.add("ddd2");
stringCollection.add("aaa2");
stringCollection.add("bbb1");
```

```
stringCollection.add("aaa1");
stringCollection.add("bbb3");
stringCollection.add("ccc");
stringCollection.add("bbb2");
stringCollection.add("ddd1");
```

Java 8中的Collections类的功能已经有所增强，你可以之直接通过调用Collections.stream()或者Collection.parallelStream()方法来创建一个流对象。下面的章节会解释这个最常用的操作。

Filter

Filter接受一个predicate接口类型的变量，并将所有流对象中的元素进行过滤。该操作是一个中间操作，因此它允许我们在返回结果的基础上再进行其他的流操作（forEach）。ForEach接受一个function接口类型的变量，用来执行对每一个元素的操作。ForEach是一个中止操作。它不返回流，所以我们不能再调用其他的流操作。

```
stringCollection
    .stream()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa2", "aaa1"
```

Sorted

Sorted是一个中间操作，能够返回一个排过序的流对象的视图。流对象中的元素会默认按照自然顺序进行排序，除非你自己指定一个Comparator接口来改变排序规则。

```
stringCollection
    .stream()
    .sorted()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa1", "aaa2"
```

一定要记住，sorted只是创建一个流对象排序的视图，而不会改变原来集合中元素的顺序。原来string集合中的元素顺序是没有改变的。

```
System.out.println(stringCollection);
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1
```

Map

map是一个对于流对象的中间操作，通过给定的方法，它能够把流对象中的每一个元素对应到另外一个对象上。下面的例子就演示了如何把每个string都转换成大写的string. 不但如此，你还可以把每一种对象映射成为其他类型。对于带泛型结果的流对象，具体的类型还要由传递给map的泛型方法来决定。

```
stringCollection
    .stream()
    .map(String::toUpperCase)
    .sorted((a, b) -> b.compareTo(a))
    .forEach(System.out::println);
// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

Match

匹配操作有多种不同的类型，都是用来判断某一种规则是否与流对象相互吻合的。所有的匹配操作都是终结操作，只返回一个boolean类型的结果。

```
boolean anyStartsWithA =
    stringCollection
        .stream()
        .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA);      // true

boolean allStartsWithA =
    stringCollection
        .stream()
        .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA);      // false

boolean noneStartsWithZ =
    stringCollection
        .stream()
        .noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ);      // true
```

Count

Count是一个终结操作，它的作用是返回一个数值，用来标识当前流对象中包含的元素数量。

```
long startsWithB =
    stringCollection
        .stream()
        .filter((s) -> s.startsWith("b"))
        .count();

System.out.println(startsWithB);    // 3
```

Reduce

该操作是一个终结操作，它能够通过某一个方法，对元素进行削减操作。该操作的结果会放在一个Optional变量里返回。

```
Optional<String> reduced =
    stringCollection
        .stream()
        .sorted()
        .reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#
ddd2"
```

Parallel Streams

像上面所说的，流操作可以是顺序的，也可以是并行的。顺序操作通过单线程执行，而并行操作则通过多线程执行。

下面的例子就演示了如何使用并行流进行操作来提高运行效率，代码非常简单。

首先我们创建一个大的list，里面的元素都是唯一的：

```
int max = 1000000;
List<String> values = new ArrayList<>(max);
for (int i = 0; i < max; i++) {
   UUID uuid = UUID.randomUUID();
   values.add(uuid.toString());
}
```

现在，我们测量一下对这个集合进行排序所使用的时间。

顺序排序

```
long t0 = System.nanoTime();

long count = values.stream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.
toMillis(t1 - t0);
System.out.println(String.format("sequential
sort took: %d ms", millis));

// sequential sort took: 899 ms
```

并行排序

```
long t0 = System.nanoTime();

long count = values.parallelStream().sorted().
count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.
toMillis(t1 - t0);
System.out.println(String.format("parallel sort
took: %d ms", millis));

// parallel sort took: 472 ms
```

如你所见，所有的代码段几乎都相同，唯一的不同就是把stream()改成了parallelStream()，结果并行排序快了50%。

Map

正如前面已经提到的那样，map是不支持流操作的。而更新后的map现在则支持多种实用的新方法，来完成常规的任务。

```
Map<Integer, String> map = new
HashMap<>();

for (int i = 0; i < 10; i++) {
   map.putIfAbsent(i, "val" + i);
}
 map.forEach((id, val) -> System.out.
println(val));
```

上面的代码风格是完全自解释的：putIfAbsent避免我们将null写入；forEach接受一个消费者对象，从而将操作实施到每一个map中的值上。

下面的这个例子展示了如何使用函数来计算map的编码

```
map.computeIfPresent(3, (num, val) -> val + num);
map.get(3);             // val33

map.computeIfPresent(9, (num, val) -> null);
map.containsKey(9);     // false

map.computeIfAbsent(23, num -> "val" + num);
map.containsKey(23);    // true

map.computeIfAbsent(3, num -> "bam");
map.get(3);             // val33
```

接下来，我们将学习，当给定一个key值时，如何把一个实例从对应的key中移除：

```
map.remove(3, "val3");
map.get(3);             // val33

map.remove(3, "val33");
map.get(3);             // null
```

另一个有用的方法：

```
map.getOrDefault(42, "not found");  // not found
```

将map中的实例合并也是非常容易的：

```
map.merge(9, "val9", (value, newValue) -> value.concat(newValue));
map.get(9);             // val9

map.merge(9, "concat", (value, newValue) -> value.concat(newValue));
map.get(9);             // val9concat
```

合并操作先看map中是否没有特定的key/value存在，如果是，则把key/value存入map，否则merging函数就会被调用，对现有的数值进行修改。

时间日期API

Java 8 包含了全新的时间日期API，这些功能都放在了java.time包下。新的时间日期API是基于Joda-Time库开发的，但是也不尽相同。下面的例子就涵盖了大多数新的API的重要部分。

Clock

Clock提供了对当前时间和日期的访问功能。Clock是对当前时区敏感的，并可用于替代System.currentTimeMillis()方法来获取当前的毫秒时间。当前时间线上的时刻可以用Instance类来表示。Instance也能够用于创建原先的java.util.Date对象。

```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();

Instant instant = clock.instant();
Date legacyDate = Date.from(instant);   // legacy java.util.Date
```

Timezones

时区类可以用一个ZoneId来表示。时区类的对象可以通过静态工厂方法方便地获取。时区类还定

义了一个偏移量，用来在当前时刻或某时间与目标时区时间之间进行转换。

```
System.out.println(ZoneId.
getAvailableZoneIds());
// prints all available timezone ids

ZoneId zone1 = ZoneId.of(“Europe/Berlin”);
ZoneId zone2 = ZoneId.of(“Brazil/East”);
System.out.println(zone1.getRules());
System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]
// ZoneRules[currentStandardOffset=-03:00]
```

LocalTime

本地时间类表示一个没有指定时区的时间，例如，10 p.m.或者17：30:15，下面的例子会用上面的例子定义的时区创建两个本地时间对象。然后我们会比较两个时间，并计算它们之间的小时和分钟的不同。

```
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2));  //
false

long hoursBetween = ChronoUnit.HOURS.
between(now1, now2);
long minutesBetween = ChronoUnit.MINUTES.
between(now1, now2);

System.out.println(hoursBetween);      // -3
System.out.println(minutesBetween);    // -239
```

LocalTime是由多个工厂方法组成，其目的是为了简化对时间对象实例的创建和操作，包括对时间字符串进行解析的操作。

```
LocalTime late = LocalTime.of(23, 59, 59);
System.out.println(late);      // 23:59:59

DateTimeFormatter germanFormatter =
   DateTimeFormatter
      .ofLocalizedTime(FormatStyle.SHORT)
      .withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse(“13:37”,
germanFormatter);
System.out.println(leetTime);   // 13:37
```

LocalDate

本地时间表示了一个独一无二的时间，例如：2014-03-11。这个时间是不可变的，与LocalTime是同源的。下面的例子演示了如何通过加减日，月，年等指标来计算新的日期。记住，每一次操作都会返回一个新的时间对象。

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1,
ChronoUnit.DAYS);
LocalDate yesterday = tomorrow.
minusDays(2);
LocalDate independenceDay = LocalDate.
```

```
of(2014, Month.JULY, 4);
DayOfWeek dayOfWeek = independenceDay.
getDayOfWeek();
System.out.println(dayOfWeek);    //
FRIDAY<span style="font-family: Georgia,
'Times New Roman', 'Bitstream Charter', Times,
serif; font-size: 13px; line-height:
19px;">Parsing a LocalDate from a string is just
as simple as parsing a LocalTime:</span>
```

解析字符串并形成LocalDate对象，这个操作和解析LocalTime一样简单。

```
DateTimeFormatter germanFormatter =
    DateTimeFormatter
        .ofLocalizedDate(FormatStyle.MEDIUM)
        .withLocale(Locale.GERMAN);

LocalDate xmas = LocalDate.
parse("24.12.2014", germanFormatter);
System.out.println(xmas);   // 2014-12-24
```

LocalDateTime

LocalDateTime表示的是日期-时间。它将刚才介绍的日期对象和时间对象结合起来，形成了一个对象实例。LocalDateTime是不可变的，与LocalTime和LocalDate的工作原理相同。我们可以通过调用方法来获取日期时间对象中特定的数据域。

```
LocalDateTime sylvester = LocalDateTime.
of(2014, Month.DECEMBER, 31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.
getDayOfWeek();
System.out.println(dayOfWeek);      //
WEDNESDAY

Month month = sylvester.getMonth();
System.out.println(month);          // DECEMBER

long minuteOfDay = sylvester.
getLong(ChronoField.MINUTE_OF_DAY);
System.out.println(minuteOfDay);    // 1439
```

如果再加上的时区信息，LocalDateTime能够被转换成Instance实例。Instance能够被转换成以前的java.util.Date对象。

```
Instant instant = sylvester
        .atZone(ZoneId.systemDefault())
        .toInstant();

Date legacyDate = Date.from(instant);
System.out.println(legacyDate);     // Wed Dec
31 23:59:59 CET 2014
```

格式化日期-时间对象就和格式化日期对象或者时间对象一样。除了使用预定义的格式以外，我们还可以创建自定义的格式化对象，然后匹配我们自定义的格式。

```
DateTimeFormatter formatter =
    DateTimeFormatter
        .ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.
parse("Nov 03, 2014 - 07:13", formatter);
```

```
String string = formatter.format(parsed);
System.out.println(string);    // Nov 03, 2014 - 07:13
```

不同于java.text.NumberFormat，新的DateTimeFormatter类是不可变的，也是线程安全的。

更多的细节，请看**这里**

Annotations

Java 8中的注解是可重复的。让我们直接深入看看例子，弄明白它是什么意思。

首先，我们定义一个包装注解，它包括了一个实际注解的数组

```
@interface Hints {
    Hint[] value();
}

@Repeatable(Hints.class)
@interface Hint {
    String value();
}
```

只要在前面加上注解名：@Repeatable，Java 8 允许我们对同一类型使用多重注解，

变体1：使用注解容器（老方法）

```
@Hints({@Hint("hint1"), @Hint("hint2")})
class Person {}
```

变体2：使用可重复注解（新方法）

```
@Hint("hint1")
@Hint("hint2")
class Person {}
```

使用变体2，Java编译器能够在内部自动对@Hint进行设置。这对于通过反射来读取注解信息来说，是非常重要的。

```
Hint hint = Person.class.getAnnotation(Hint.class);
System.out.println(hint);                   // null

 Hints hints1 = Person.class.getAnnotation(Hints.class);
System.out.println(hints1.value().length);  // 2

Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class);
System.out.println(hints2.length);          // 2
```

尽管我们绝对不会在Person类上声明@Hints注解，但是它的信息仍然可以通过getAnnotation(Hints.class)来读取。并且，getAnnotationsByType方法会更方便，因为它赋予了所有@Hints注解标注的方法直接的访问权限。

```
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@interface MyAnnotation {}
```

先到这里

我的Java 8编程指南就到此告一段落。当然，还有很多内容需要进一步研究和说明。这就需要靠读者您来对JDK 8进行探究了，例如：Arrays.parallelSort，StampedLock和CompletableFuture等等 ———— 我这里只是举几个例子而已。■

《开发月刊》电子杂志

51CTO开发频道

数据